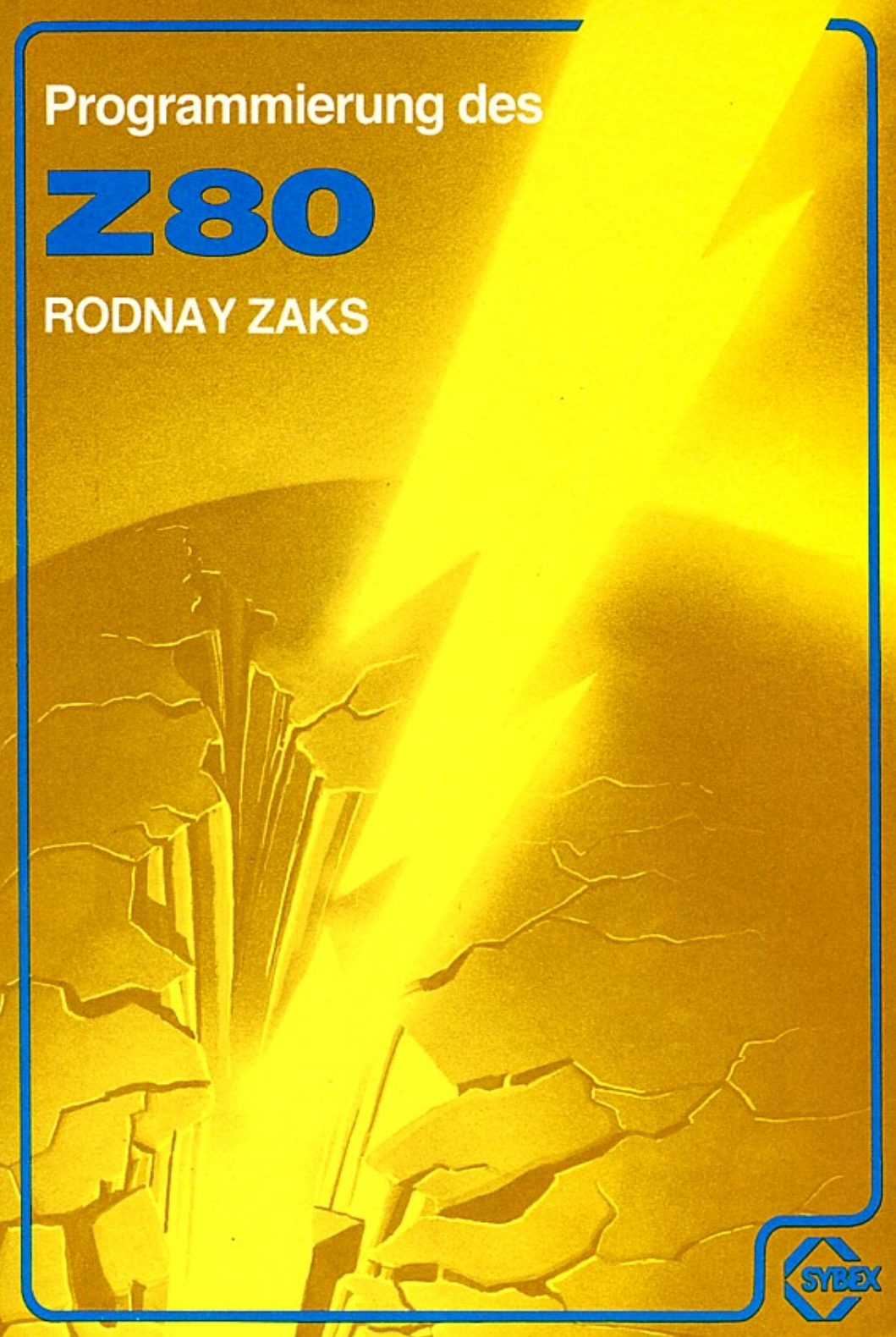
Programmierung des

Z80

RODNAY ZAKS



SYBEX

Programmierung des Z80

Programmierung des Z80

Rodnay Zaks

Anmerkung:
Z80 ist ein geschütztes Warenzeichen von ZILOG Inc., USA.

Originalausgabe in Englisch
Titel der englischen Ausgabe: Programming the Z80
Original Copyright © 1980 by SYBEX Inc., Berkeley, USA

Deutsche Übersetzung: Bernd Ploss

Umschlag: Daniel Le Noury
Satz: tgr typo-grafik-repro gmbH., Remscheid
Gesamtherstellung: Druckerei Hub. Hoch, Düsseldorf

Der Verlag hat alle Sorgfalt walten lassen, um vollständige und akkurate Informationen zu publizieren. SYBEX-Verlag GmbH, Düsseldorf, übernimmt keine Verantwortung für die Nutzung dieser Informationen, auch nicht für die Verletzung von Patent- und anderen Rechten Dritter, die daraus resultieren. Hersteller behalten das Recht, Schaltpläne und technische Charakteristika ohne Bekanntgabe an die Öffentlichkeit zu ändern. Für genaue technische Daten auf dem neuesten Stand wird der Leser an die Hersteller verwiesen.

ISBN 3-88745-006-X
1. Auflage 1982
2. Auflage 1982
3. Auflage 1983
4. Auflage 1983
5. Auflage 1984
6. Auflage 1984

Alle deutschsprachigen Rechte vorbehalten. Kein Teil des Werkes darf in irgendeiner Form (Druck, Fotokopie, Mikrofilm oder einem anderen Verfahren) ohne schriftliche Genehmigung des Verlages reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.

Printed in Germany
Copyright © 1982, SYBEX-Verlag GmbH., Düsseldorf

Dankwort

Der Entwurf eines Textbuches über Programmierung ist immer schwierig. Ein solches Buch aber so zu konzipieren, daß es als Einführung in die Programmierung taugt und trotzdem fortgeschrittene Konzepte präsentiert, ohne daß Hardware- und Softwareaspekte außer acht gelassen werden, ist eine Herausforderung. Der Autor möchte sich hier für die vielen Vorschläge zu Verbesserungen und Änderungen des Buches bei O. M. Barlow, Dennis L. Feick, Richard D. Reid, Stanley E. Erwin, Philip Hooper, Dennis B. Kitsz, R. Ratke und Jim Crocker bedanken. Sein besonderer Dank gilt Chris Williams für seine Beiträge zu dem Z80 Instruktions-Set und dem Kapitel über Datenstrukturen.

Alle weiteren Vorschläge zur Verbesserung oder Änderung des Buches sollten an den Verlag geschickt werden.

Die Tabellen zu den Hexadezimal-Kodes der Z80-Instruktionen in Kapitel 4 sind mit Genehmigung von Zilog Inc, USA, nachgedruckt. Tabellen 2.26 und 2.27 wurden mit Genehmigung von Intel Corporation, USA, abgedruckt.

Inhaltsverzeichnis

I. Grundbegriffe 17
Einführung, Was ist Programmierung?, Flußdiagramme, Darstellung von Information

II. Z80 Hardware Organisation 43
Einführung, Systemarchitektur, Interne Organisation des Z80, Befehlsformate, Befehlsausführung durch den Z80, Zusammenfassung

III. Grundlegende Techniken der Programmierung 89
Einführung, Arithmetische Programme, Multiplikation von BCD-Zahlen, Division von Dualzahlen, Zusammengefaßte Darstellung der Befehle, Unterprogramme, Zusammenfassung

IV. Der Befehlssatz des Z80 147
Einführung, Klassen von Befehlen, Zusammenfassung, Beschreibung der einzelnen Befehle

V. Adressierungstechniken 429
Einführung, Mögliche Arten der Adressierung, Die Adressierungsformen des Z80, Der Gebrauch der Z80-Adressierungsformen, Zusammenfassung

VI. Ein-/Ausgabetechniken 449
Einführung, Eingabe und Ausgabe, Parallele wortweise Übertragung, Serielle bitweise Übertragung, Zusammengefaßte Darstellung der Peripherie, Verwaltung von Ein- und Ausgabe, Zusammenfassung

VII. Ein-/Ausgabegeräte 497
Einführung, Die Standard-PIO, Das interne Steuerregister, Programmierung einer PIO, Die Zilog-Z80-PIO

VIII. Beispiel für Anwendungen 507
Einführung, Löschen eines Speicherbereichs, Abfrage von Ein-/Ausgabegeräten, Eingabe von Zeichen, Test eines Zeichens, Test eines Zeichenbereiches, Erzeugen des Paritybits, Codeumwandlung: ASCII nach BCD, Hexadezimal nach ASCII, Suchen des größten Elementes in einer Tabelle, Summe von n Elementen, Berechnung einer Prüfsumme, Zählen der Nullen, Übertragung von Blöcken, Transfer eines Blocks von BCD-Zahlen, Vergleich zweier vorzeichenbehafteter 16-Bit-Zahlen, Bubble-Sort, Zusammenfassung

IX. Datenstrukturen 525
1. Teil – Theorie
Einführung, Zeiger, Listen, Suchen und Sortieren, Zusammenfassung
2. Teil – Beispiele zum Entwurf
Einführung, Darstellung der Daten in der Liste, Eine einfache Liste, Alphabetische Liste, Verkettete Liste, Zusammenfassung

X. Entwicklung von Programmen 563
Einführung, Grundlegende Entscheidungen, Unterstützende Software, Wie man schrittweise ein Programm entwickelt, Alternativen bei der Hardware, Der Assembler, Bedingte Assemblierung, Zusammenfassung

XI. Schluß 585
Die technologische Entwicklung, Der nächste Schritt

Anhang A 587
Tabelle zur Umwandlung von Hexadezimalzahlen

Anhang B 588
ASCII-Tabelle

Anhang C 589
Tabelle relativer Sprungdistanzen

Anhang D 590
Umwandlung dezimal nach BCD

Anhang E 591
Der Befehlssatz des Z80

Anhang F 598
Äquivalente Befehle: Z80–8080

Anhang G 599
Äquivalente Befehle: 8080–Z80

Index 601

Verzeichnis der Abbildungen

Abb. 1.1: Flußdiagramm zur Konstanthaltung der Raumtemperatur . . . 19
Abb. 1.2: Dezimal-Dual-Tabelle . . . 22
Abb. 1.3: Tabelle der Zweierkomplemente . . . 28
Abb. 1.4: BCD Tabelle . . . 34
Abb. 1.5: Typische Gleitkommadarstellung . . . 36
Abb. 1.6: ASCII-Umwandlungs-Tabelle . . . 38
Abb. 1.7: Oktale Symbole . . . 39
Abb. 1.8: Hexadezimalkode . . . 41
Abb. 2.1: Z80-Standardsystem . . . 44
Abb. 2.2: „Standardarchitektur" eines Mikroprozessors . . . 46
Abb. 2.3: Schieben und Rotieren . . . 47
Abb. 2.4: Die 16-Bit-Adreßregister erzeugen den Adreßbus . . 48
Abb. 2.5: Die beiden Befehle zur Manipulation des Stapels . . 51
Abb. 2.6: Einen Befehl aus dem Speicher holen . . . 52
Abb. 2.7: Automatische Abfolge . . . 52
Abb. 2.8: Architektur mit einem einzelnen Bus . . . 53
Abb. 2.9: Ausführung einer Addition – R0 in den Akku . . . 54
Abb. 2.10: Addition – Zweites Register R1 in die ALU . . . 54
Abb. 2.11: Das Ergebnis wird erzeugt und kommt nach R0 . . . 55
Abb. 2.12: Der kritische Wettlauf . . . 56
Abb. 2.13: Zwei Puffer werden benötigt . . . 56
Abb. 2.14: Interne Organisation des Z80 . . . 61
Abb. 2.15: Typisches Befehlsformat . . . 63
Abb. 2.16: Die Kodes der Register . . . 64
Abb. 2.17: Instruction Fetch – (PC) wird zum Speicher geschickt 66
A b. 2.18: PC wird inkrementiert . . . 66
Abb. 2.19: Der Befehl kommt aus dem Speicher ins IR . . . 67
Abb. 2.20: Übertragung von C nach D . . . 68
Abb. 2.21: Der Inhalt von C wird ins TMP abgelegt . . . 68
Abb. 2.22: Der Inhalt von TMP wird nach D übertragen . . . 69
Abb. 2.23: Zwei Transfers werden gleichzeitig ausgeführt . . . 71
Abb. 2.24: Das Ende von ADD . . . 72
Abb. 2.25: Fetch-Execute-Überlappung während T1–T2 . . . 73
Abb. 2.26: Abkürzungen von INTEL . . . 74
Abb. 2.27: Intel Befehlsformate . . . 76
Abb. 2.28: Übertragung des Inhalts von HL zum Adreßbus . . . 80
Abb. 2.29: LD A,(Adresse) ist ein Dreiwortbefehl . . . 81
Abb. 2.30: Vor der Ausführung von LD A . . . 81
Abb. 2.31: Nach der Ausführung von LD A . . . 82
Abb. 2.32: Das zweite Byte des Befehls gelangt nach Z . . . 82
Abb. 2.33: Anschlußbelegung der Z80 MPU . . . 85

Abb. 3.0: Die Register des Z80 90
Abb. 3.1: Acht-Bit-Addition RES = OP1 + OP2 91
Abb. 3.2: LD A,(ADR1): OP1 wird aus dem Speicher geladen 92
Abb. 3.3: ADD A,(HL) 92
Abb. 3.4: LD (ADR3),A (Lege den Akkumulator im Speicher ab) 93
Abb. 3.5: 16-Bit-Addition – die Operanden 94
Abb. 3.6: Speicherung der Operanden in umgekehrter Reihenfolge 96
Abb. 3.7: Zeiger auf das obere Byte 97
Abb. 3.8: Eine 32-Bit-Addition 98
Abb. 3.9: 16-Bit-Ladebefehl – LD HL,(ADR1) 99
Abb. 3.10: Speicherung von BCD-Ziffern 102
Abb. 3.11: Gepackte BCD-Subtraktion: N1←N2 – N1 105
Abb. 3.12: Der grundlegende Algorithmus für die Multiplikation – ein Flußdiagramm 107
Abb. 3.13: 8 x 8 Bit Multiplikationsprogramm 108
Abb. 3.14: 8 x 8 Bit Multiplikation – die Register 109
Abb. 3.15: LD BC,(MPRAD) 110
Abb. 3.16: LD DE,(MPDAD) 111
Abb. 3.17: Schieben und Rotieren 113
Abb. 3.18: Von E nach D Schieben 114
Abb. 3.19: Tabelle zur Multiplikationsaufgabe 115
Abb. 3.20: Multiplikation: Nach dem ersten Befehl 116
Abb. 3.21: Multiplikation: Nach zwei Befehlen 117
Abb. 3.22: Multiplikation: Nach fünf Befehlen 117
Abb. 3.23: Ein Durchlauf durch die Schleife 118
Abb. 3.24: Verbesserte Multiplikation, Schritt 1 119
Abb. 3.25: Register zur verbesserten Multiplikation 120
Abb. 3.26: Verbesserte Multiplikation, Schritt 2 121
Abb. 3.27: 16 x 16 Bit Multiplikation – die Register 122
Abb. 3.28: 16 x 16 Bit Multiplikationsprogramm 123
Abb. 3.29: 16 x 16 Bit Multiplikation mit 32-Bit-Ergebnis . . . 125
Abb. 3.30: Flußdiagramm für duale 8-Bit-Division 126
Abb. 3.31: 16 / 8 Bit Division – die Register 126
Abb. 3.32: 16 / 8 Bit Divisionsprogramm 127
Abb. 3.33: Formular für das Divisionsprogramm 129
Abb. 3.34: Nicht-wiederherstellende Division – die Register 130
Abb. 3.35: Unterprogrammaufrufe 135
Abb. 3.36: Verschachtelte Aufrufe 137
Abb. 3.37: Die Unterprogrammaufrufe 138
Abb. 3.38: Der Stapel zu verschiedenen Zeitpunkten 138
Abb. 3.39: Multiplikation: eine vollständige Aufzeichnung . . . 143
Abb. 3.40: Das Multiplikationsprogramm (hexadezimal) . . . 145
Abb. 3.41: Zwei Durchläufe durch die Schleife 145
Abb. 4.1: Schieben und Rotieren 148

Abb. 4.2: Acht-Bit-Ladebefehle – LD 152
Abb. 4.3: 16-Bit-Ladebefehle – 'LD', 'PUSH' und 'POP' . . . 153
Abb. 4.4: Austausch 'EX' und 'EXX' 154
Abb. 4.5: Blocktransferbefehle 155
Abb. 4.6: Blocksuchbefehle 156
Abb. 4.7: Acht-Bit Arithmetik und Logik 157
Abb. 4.8: 16-Bit Arithmetik und Logik 158
Abb. 4.9: Schieben und Rotieren 161
Abb. 4.10: Rotieren und Schieben 161
Abb. 4.11: Neun-Bit-Rotation 162
Abb. 4.12: Acht-Bit-Rotation 162
Abb. 4.13: Rotieren von Ziffern 163
Abb. 4.14: Bitmanipulationen 164
Abb. 4.15: Universelle Operationen mit AF 165
Abb. 4.16: Das Flagregister 165
Abb. 4.17: Zusammenfassung der Arbeitsweise der Flags . . . 170
Abb. 4.18: Sprungbefehle 172
Abb. 4.19: Restart-Befehle 173
Abb. 4.20: Die Ausgabebefehle 175
Abb. 4.21: Die Eingabebefehle 176
Abb. 4.22: Verschiedene CPU Steuerbefehle 177
Abb. 5.1: Grundlegende Adressierungsarten 430
Abb. 5.2: Adressierung (direkt indiziert) 433
Abb. 5.3: Indirekte indizierte Adressierung 433
Abb. 5.4: Indirekte Adressierung 434
Abb. 5.5: Die indizierte Adressierung hat einen Zweibyte-Opcode 438
Abb. 5.6: Flußdiagramm zur Zeichensuche 440
Abb. 5.7: Blocktransfer – Initialisierung der Register 441
Abb. 5.8: Speicherbelegung beim Blocktransfer 443
Abb. 5.9: Addition zweier Blöcke: BLK1=BLK1+BLK2 . . . 446
Abb. 5.10: Speicherorganisation für den Blocktransfer 447
Abb. 6.1: Ein Relais einschalten 451
Abb. 6.2: Ein programmierter Impuls 451
Abb. 6.3: Prinzipielles Flußdiagramm einer Verzögerungsschleife 452
Abb. 6.4: Parallele Übertragung von Worten – der Speicher 456
Abb. 6.5: Parallele Übertragung von Worten: Flußdiagramm 457
Abb. 6.6: Bitserielle Übertragung 461
Abb. 6.7: Seriell nach Parallel: die Register 462
Abb. 6.8: Handshaking (Ausgabe) 466
Abb. 6.8a: Handshaking (Eingabe) 466
Abb. 6.9: Drucker – Datenwege 467
Abb. 6.10: Siebensegmentanzeige LED 469
Abb. 6.11: Hexadezimalzeichen, die mit einer Siebensegmentanzeige erzeugt werden 469

Abb. 6.12: Format eines Fernschreiberwortes 472
Abb. 6.13: TTY-Eingabe mit Echo 473
Abb. 6.14: Fernschreiberprogramm 474
Abb. 6.15: Eingabe vom Fernschreiber 475
Abb. 6.16: Ausgabe auf den Fernschreiber 476
Abb. 6.17: Ausdruck eines Speicherbereichs 478
Abb. 6.18: Drei Methoden der Ein-/Ausgabesteuerung 479
Abb. 6.19: Flußdiagramm einer Pollingschleife 481
Abb. 6.20: Eingabe von einem Lochstreifenleser 481
Abb. 6.21: Ausgabe auf einen Stanzer oder Drucker 482
Abb. 6.22: Der Stapel des Z80 nach einem Interrupt 482
Abb. 6.23: Sicherstellen einiger Arbeitsregister 483
Abb. 6.24: Interrupt Folge 484
Abb. 6.25: Der NMI erzwingt eine automatische Verzweigung 485
Abb. 6.26: Interrupt Modi 487
Abb. 6.27: Sicherstellen der Register 488
Abb. 6.28: Interrupt Modus 1 489
Abb. 6.29: Interrupt Modus 2 490
Abb. 6.30: Modus 2 – ein praktisches Beispiel 491
Abb. 6.31: Abfrage und vektorisierter Interrupt 493
Abb. 6.32: Mehrere Geräte können die gleiche Interruptleitung verwenden 494
Abb. 6.33: Inhalt des Stapels während mehrerer Interrupts . . . 494
Abb. 6.34: Interruptlogik 496
Abb. 7.1: Typische PIO 498
Abb. 7.2: Der Gebrauch einer PIO – Laden des Kontrollregisters 499
Abb. 7.3: Der Gebrauch einer PIO – Laden der Datenrichtung 500
Abb. 7.4: Der Gebrauch einer PIO – Lesen des Status 500
Abb. 7.5: Der Gebrauch einer PIO – Lesen einer Eingabe 501
Abb. 7.6: Die Anschlußbelegung der Z80 PIO 502
Abb. 7.7: Blockdiagramm der Z80 PIO 503
Abb. 8.1: Größtes Element in einer Tabelle 513
Abb. 8.2: Summe von N Elementen 514
Abb. 8.3: BCD-Blocktransfer – der Speicher 517
Abb. 8.4: Vergleich zweier vorzeichenbehafteter Zahlen . . . 518
Abb. 8.5: Bubble-Sort Beispiel: Schritte 1 bis 12 520
Abb. 8.6: Bubble-Sort Beispiel: Schritte 13 bis 21 521
Abb. 8.7: Bubble-Sort Flußdiagramm 522
Abb. 8.8: Bubble-Sort 523
Abb. 9.1: Ein indirekter Zeiger 526
Abb. 9.2: Struktur eines Inhaltsverzeichnisses 527
Abb. 9.3: Eine verkettete Liste 528
Abb. 9.4: Einfügen eines Blocks 528
Abb. 9.5: Eine Schlange 529
Abb. 9.6: Ringliste 530

Abb. 9.7: Stammbaum 531
Abb. 9.8: Doppelt verkettete Liste 531
Abb. 9.9: Die Tabellenstruktur 534
Abb. 9.10: Typische Eintragungen in die Liste 534
Abb. 9.11: Die einfache Liste 535
Abb. 9.12: Flußdiagramm zur Suche in der Tabelle 536
Abb. 9.13: Flußdiagramm zum Einfügen in die Tabelle 537
Abb. 9.14: Löschen einer Eintragung (einfache Liste) 538
Abb. 9.15: Löschen in der Tabelle – Flußdiagramm 539
Abb. 9.16: Einfache Liste – die Programme 540
Abb. 9.17: Einfache Liste – ein Durchlauf als Beispiel 541
Abb. 9.18: Binäres Suchen – Flußdiagramm 544
Abb. 9.19: Eine binäre Suche 546
Abb. 9.20: Füge „BAC“ ein 548
Abb. 9.21: Lösche „BAC“ 548
Abb. 9.22: Flußdiagramm zum Löschen (alphabetische Liste) 549
Abb. 9.23: Binäres Suchprogramm 550
Abb. 9.24: Alphabetische Liste – Ein Lauf als Beispiel 553
Abb. 9.25: Struktur der verketteten Liste 555
Abb. 9.26: Verkettete Liste – Eine Suche 557
Abb. 9.27: Verkettete Liste: Beispiel des Einfügens 557
Abb. 9.28: Beispiel des Löschens (Verkettete Liste) 558
Abb. 9.29: Verkettete Liste – die Programme 559
Abb. 9.30: Verkettete Liste – ein Durchlauf 561
Abb. 10.1: Programmierebenen 565
Abb. 10.2: Ein typischer Speicherbelegungsplan 570
Abb. 10.3: Programmierformular für einen Mikroprozessor 575
Abb. 10.4: Assemblerliste – ein Beispiel 576
Abb. 10.5: Priorität der Operatoren 580

Vorwort

Dieses Buch wurde konzipiert als ein vollständiger in sich abgeschlossener Text zum Erlernen des Programmierens unter Verwendung des Z80. Es kann von allen gelesen werden, die auch bisher noch nicht programmiert haben. Von besonderem Wert sollte es für alle diejenigen sein, die den Z80 verwenden.

Für alle Leser, die schon programmiert haben, lehrt dieses Buch spezielle Programmiertechniken unter Verwendung der besonderen Eigenschaften des Z80. Dieser Text deckt die Methoden elementarer bis mittlerer Schwierigkeit ab, die man braucht, um effektiv zu programmieren.

Ziel dieses Textes ist es, demjenigen, der diesen Mikroprozessor verwenden will, fundierte Grundlagen zu liefern. Natürlich lehrt kein Buch wirklich zu programmieren, solange man es nicht tatsächlich tut. Es ist jedoch zu hoffen, daß dieses Buch den Leser zu einem Punkt führt, an dem er sich in der Lage fühlt, mit dem Programmieren selbst zu beginnen, und an dem er einfache oder auch etwas anspruchsvollere Probleme mit einem Mikroprozessor lösen kann.

Dieses Buch basiert auf der Erfahrung des Autors, der mehr als 1000 Personen gelehrt hat, wie man Mikrocomputer programmiert. Deshalb ist es konsequent strukturiert. Jedes Kapitel führt normalerweise von einfachen zu komplexeren Problemen. Leser, die die elementare Programmierung schon erlernt haben, können den Anfang eines Kapitels überspringen. Andere, die noch nie programmiert haben, müssen vielleicht die hinteren Abschnitte jedes Kapitels ein zweites Mal durcharbeiten. Das Buch wurde dazu entworfen, den Leser durch sämtliche grundlegende Konzepte und Techniken zu führen, die man für Programme wachsender Komplexität braucht. Deshalb sei es dringend empfohlen, die Reihenfolge der Kapitel einzuhalten. Um wirkliche Erfolge zu erzielen, ist es zusätzlich wichtig, daß der Leser versucht, so viele Aufgaben wie möglich zu lösen. Die Schwierigkeit der Aufgaben wurde sorgfältig abgestimmt. Sie dienen der Überprüfung, daß der behandelte Stoff auch wirklich verstanden wurde. Wenn man die Aufgaben nicht bearbeitet, dann kann man den Wert dieses Buchs als Lehrmittel nicht voll nutzen. Mehrere der Aufgaben können z. T. erhebliche Zeit beanspruchen, wie z. B. die Multiplikationsübung. Löst man jedoch auch diese Aufgaben, so lernt man Programmieren wirklich. Dies ist unerläßlich.

Für diejenigen, die am Ende dieses Buchs am Programmieren Geschmack gefunden haben, sei das Buch „Z80 Anwendungen“ von James W. Coffron (Ref.-Nr. 3037, August 1984) empfohlen.

Andere Bücher in dieser Serie behandeln die Programmierung anderer weit verbreiteter Mikroprozessoren.

Für diejenigen, die ihre Hardwarekenntnisse erweitern wollen, seien die Bücher „Chip und System: Einführung in die Mikroprozessoren-Technik" (Ref.-Nr.: 3017) und „Mikroprozessor Interface Techniken" (Ref.-Nr.: 3012) empfohlen.

Der Inhalt dieses Buchs wurde sorgfältig geprüft und sollte korrekt sein. Jedoch können trotzdem einige unvermeidliche Schreibfehler oder andere Irrtümer enthalten sein. Der Autor ist für alle Hinweise aufmerksamer Leser dankbar, so daß zukünftige Ausgaben von Ihren Erfahrungen profitieren können. Auch für andere Verbesserungsvorschläge (z. B. Programme, die von Lesern gewünscht, entwickelt oder für wichtig befunden werden) ist der Autor dankbar.

1

Grundbegriffe

Einführung

Dieses Kapitel soll in die Grundbegriffe und Bezeichnungen der Computerprogrammierung einführen. Der Leser, der mit diesen Begriffen schon vertaut ist, wird vielleicht den Inhalt dieses Kapitels nur flüchtig durchsehen und sich dann Kapitel 2 zuwenden. Es sei jedoch auch dem erfahrenen Leser empfohlen, den Inhalt dieses einführenden Kapitels genau anzusehen. Viele wichtige Begriffe werden hier erläutert, z. B. Zweierkomplement, BCD und andere Darstellungen. Einige dieser Begriffe sind für den Leser vielleicht neu, andere erweitern die Kenntnisse und Fähigkeiten auch des erfahrenen Programmierers.

Was ist Programmierung?

Ist ein Problem gegeben, muß man zuerst eine Lösung finden. Diese Lösung in Form eines schrittweisen Verfahrens nennt man einen *Algorithmus*. Ein Algorithmus ist eine Vorschrift, nach der ein gegebenes Problem Schritt für Schritt gelöst wird. Er muß nach einer endlichen Zahl von Schritten zum Abschluß gebracht werden. Den Algorithmus kann man in einer beliebigen Sprache oder Symbolik darstellen. Ein einfaches Beispiel für einen Algorithmus ist:

1. Stecke den Schlüssel in das Schlüsselloch
2. Drehe den Schlüssel eine Umdrehung links herum
3. Fasse den Türdrücker an
4. Bewege den Türdrücker nach unten und drücke gegen die Tür.

Stimmt der Algorithmus für den vorliegenden Typ von Türschloß, wird sich die Tür jetzt öffnen. Diese Vorschrift aus vier Schritten beschreiben einen Algorithmus zum Öffnen einer Tür.

Sobald die Lösung für ein Problem in Form eines Algorithmus dargestellt ist, kann der Algorithmus von dem Computer abgearbeitet werden. Leider ist es eine feststehende Tatsache, daß Computer normales gesprochenes Deutsch (oder jede andere natürliche Sprache) weder verstehen noch ausführen können. Der Grund ist die *syntaktische Mehrdeutigkeit* aller natürlicher Sprachen. Nur eine wohldefinierte Teilmenge einer natürlichen Sprache kann von einem Computer „verstanden" werden. Diese nennt man dann eine *Programmiersprache*.

Einen Algorithmus in eine Folge von Befehlen in einer Programmiersprache zu übersetzen, nennt man *Programmierung*. Genauer ausgedrückt heißt die Übersetzung des Algorithmus in die Programmiersprache *Kodierung*. Programmierung bezieht sich in Wirklichkeit nicht nur auf die Kodierung, sondern auf den gesamten Entwurf der Programme und der „Datenstrukturen", die von dem Algorithmus benutzt werden.

Effektive Programmierung setzt nicht nur voraus, daß man verschiedene Techniken zur Ausführung von Standardalgorithmen versteht, sondern auch daß man alle Hardwarehilfsmittel des Computers, wie interne Register, Speicher und periphere Geräte, geschickt nutzt und die passenden Datenstrukturen kreativ einsetzt. Diese Techniken werden in den nächsten Kapiteln behandelt.

Programmierung verlangt auch eine konsequente Disziplin bei der Dokumentation, so daß die Programme für andere wie auch für den Autor verständlich sind. Zu einem Programm gehört sowohl eine interne als auch eine externe Dokumentation.

Interne Dokumentation bezieht sich auf Kommentare innerhalb des Programms, die sein Funktionieren erklären. Externe Dokumentation sind Unterlagen zum Entwurf, die vom Programm getrennt sind, wie schriftliche Erklärungen, Handbücher und Flußdiagramme.

Flußdiagramme

Fast immer wird zwischen *Algorithmus* und *Programm* ein Zwischenschritt eingeschaltet, das *Flußdiagramm*. In einem Flußdiagramm wird der Algorithmus einfach als Folge von Rechtecken und Rauten, die die einzelnen Schritte des Algorithmus enthalten, symbolisch dargestellt. Rechtecke verwendet man für *Befehle* oder „ausführbare Anweisungen", Rauten für *Verzweigungen* wie: Ist die Aussage X wahr, dann gehe nach A, sonst nach B. Statt jetzt eine formale Beschreibung von Flußdiagrammen zu bringen, werden wir Flußdiagramme später in diesem Buch einführen und diskutieren, wenn wir Programme erläutern.

Flußdiagramme werden dringend empfohlen als Zwischenschritt zwischen der Spezifikation des Algorithmus und der Kodierung der Lösung! Bemerkenswerterweise hat man herausgefunden, daß etwa 10% der Programmierer ein Programm erfolgreich schreiben können, ohne ein Flußdiagramm zu haben. Man hat aber auch herausgefunden, daß unglücklicherweise 90% der Programmierer meinen, sie gehörten zu diesen 10%! Das Ergebnis: Durchschnittlich 80% der Programme versagen, wenn sie zum ersten Mal auf einen Computer laufen sollen. (Die Prozentangaben sind natürlich nicht exakt.) Kurz gesagt, die meisten Programmiererneulinge sehen die Notwendigkeit, ein Flußdiagramm zu zeichnen, nur selten ein. Dies führt üblicherweise zu „unsauberen" oder fehlerhaften Programmen. Sie verbrauchen dann viel Zeit beim Testen und Korrigieren ihrer Programme (das nennt man dann *Debugging*). Flußdiagramme zu zeichnen, sei deshalb in allen Fällen dringend emp-

fohlen. Dazu braucht man vor der Kodierung nur wenig zusätzliche Zeit, erhält aber normalerweise ein klares Programm, das richtig und schnell funktioniert. Sind die Flußdiagramme einmal gut verstanden, dann wird ein kleiner Prozentsatz der Programmierer in der Lage sein, diesen Schritt im Geiste auszuführen, ohne ihn zu Papier zu bringen. Leider sind dann die Programme, die sie schreiben, für andere meist nur schwer zu verstehen, weil die Dokumentation durch die Flußdiagramme fehlt. Deshalb wird allgemein empfohlen, das Verfahren der Flußdiagramme bei allen wichtigen Programmen konsequent anzuwenden. Viele Beispiele werden das ganze Buch hindurch vorgestellt werden.

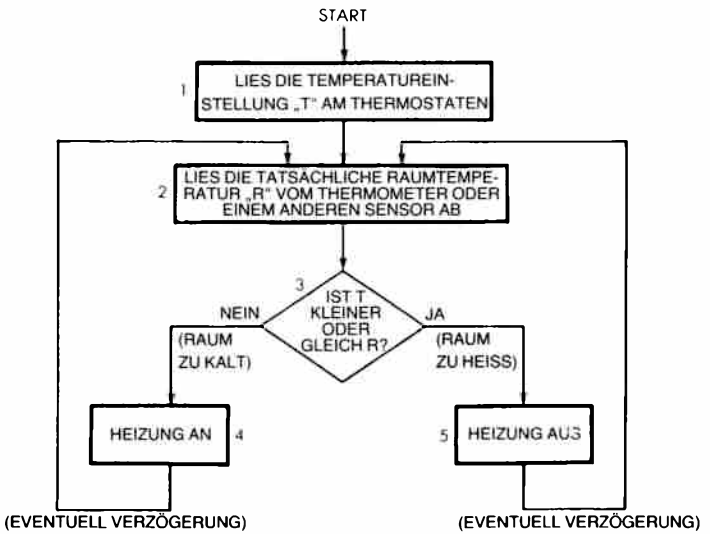


Abb. 1.1: Flußdiagramm zur Konstanthaltung der Raumtemperatur

Darstellung von Information

Alle Computer handhaben Informationen in Form von Zahlen oder in Form von Zeichen. Wir wollen hier die externe und die interne Darstellung von Information in einem Computer untersuchen.

Interne Darstellung von Information

In einem Computer wird jede Information als Gruppe von Bits dargestellt. Bit ist die Abkürzung von binary digit, d. h. *Binärziffer* („0" oder „1"). Eingeschränkt durch die Eigenschaften der üblichen Elektronik, verwendet die einzige praktikable Darstellung von Information eine Logik aus zwei Zuständen (die Darstellung der Zustände „0" und „1"). Die zwei Zustände der Schaltungen, die in der digitalen Elektronik verwendet werden, sind allgemein „ein" und „aus", und diese werden durch die

Symbole „0" und „1" logisch dargestellt. Weil diese Schaltungen verwendet werden, um „logische" Funktionen auszuführen, nennt man sie „binäre Logik". Als Ergebnis davon wird heute tatsächlich die gesamte Informationsverarbeitung in binärem Format ausgeführt. Bei den Mikroprozessoren im allgemeinen und beim Z80 im besonderen sind diese Bits in Achtergruppen strukturiert. Eine Gruppe von acht Bit nennt man ein *Byte*, eine Gruppe von vier Bit nennt man ein *Nibble*.

Wir wollen jetzt untersuchen, wie Information intern im Binärformat dargestellt wird. Zwei Bereiche müssen dabei innerhalb des Rechners unterschieden werden. Der erste ist das Programm, d. h. eine Folge von Befehlen. Der Zweite sind die Daten, mit denen das Programm arbeitet, und die Zahlen oder alphanumerischen Text enthalten können. Wir werden jetzt folgende drei Darstelungen von Information diskutieren: Programme, Zahlen und alphanumerische Daten.

Darstellung von Programmen

Alle Befehle werden intern durch ein oder mehrere Byte dargestellt. Ein sogenannter „Kurzbefehl" besteht aus einem einzelnen Byte. Ein längerer Befehl besteht aus zwei oder mehr Bytes. Da der Z80 ein Acht-Bit-Mikroprozessor ist, holt er Byte für Byte nacheinander aus dem Speicher. Deshalb kann ein Ein-Byte-Befehl immer schneller ausgeführt werden, als ein Zwei- oder Drei-Byte-Befehl. Wir werden später sehen, daß dies ein entscheidendes Merkmal für den Befehlssatz jedes Mikroprozessors und damit auch des Z80 ist, bei dem besonderer Aufwand getrieben wurde, möglichst viele Ein-Byte-Befehle zur Verfügung zu stellen, um die Leistungsfähigkeit bei der Programmausführung zu verbessern. Die Begrenzung auf eine Länge von acht Bit hat jedoch zu wichtigen Einschränkungen geführt, die später herausgestellt werden sollen. Dies ist ein klassisches Beispiel für den Kompromiß zwischen Geschwindigkeit und Flexibilität bei der Programmierung. Der binäre Code zur Darstellung der Befehle ist vom Hersteller vorgeschrieben. Wie jeder andere Mikroprozessor wird der Z80 mit einem festen Befehlssatz geliefert. Diese Befehle sind vom Hersteller definiert und werden am Ende des Buches mit dem zugehörigen Code aufgelistet. Jedes Programm wird durch eine Folge dieser binären Befehle gebildet. Die Z80-Befehle werden in Kapitel 4 vorgestellt.

Darstellung numerischer Daten

Die Darstellung von Zahlen ist nicht ganz einfach und man muß verschiedene Fälle unterscheiden. Zuerst müssen wir ganze, dann vorzeichenbehaftete, d. h. positive und negative Zahlen, und schließlich Dezimalzahlen darstellen können. Jetzt wollen wir diese Bedingungen und mögliche Lösungen ansprechen.

Ganze Zahlen kann man dual darstellen. Die duale Form ist einfach die Darstellung des Zahlenwertes im Dualsystem. Im Dualsystem repräsen-

tiert das Bit ganz rechts 2 hoch 0. Das nächste Bit links repräsentiert 2 hoch 1, das nächste 2 hoch 2 und das Bit ganz links 2 hoch 7 = 128.

$b_7b_6b_5b_4b_3b_2b_1b_0$ bedeutet
$b_7 2^7 + b_6 2^6 + b_5 2^5 + b_4 2^4 + b_3 2^3 + b_2 2^2 + b_1 2^1 + b_0 2^0$

Die Potenzen von 2 sind:

$2^7=128, 2^6=64, 2^5=32, 2^4=16, 2^3=8, 2^2=4, 2^1=2, 2^0=1$

Die duale Darstellung entspricht der dezimalen Zahlendarstellung, in der z. B. „123" bedeutet:

```
  1 x 100 = 100
+ 2 x  10 =  20
+ 3 x   1 =   3
-------------------
        = 123
```

Beachten Sie, daß $100=10^2$, $10=10^1$ und $1=10^0$.

In diesem „Stellenwertsystem" bedeutet jede Ziffer eine Zehnerpotenz. Im Dualsystem repräsentiert jede Binärziffer oder jedes „Bit" eine Zweierpotenz statt einer Zehnerpotenz im Dezimalsystem.

Beispiel:„ 00001001" dual bedeutet:

```
1 x   1 = 1    (2^0)
0 x   2 = 0    (2^1)
0 x   4 = 0    (2^2)
1 x   8 = 8    (2^3)
0 x  16 = 0    (2^4)
0 x  32 = 0    (2^5)
0 x  64 = 0    (2^6)
0 x 128 = 0    (2^7)
-------------------
dezimal     9
```

Wir wollen einige weitere Beispiele untersuchen:

„10000001" bedeutet:

```
1 x   1 =   1
0 x   2 =   0
0 x   4 =   0
0 x   8 =   0
0 x  16 =   0
0 x  32 =   0
0 x  64 =   0
1 x 128 = 128
-------------------
dezimal     129
```

„10000001" bedeutet also dezimal 129.

Wenn Sie die duale Zahlendarstellung untersuchen, dann werden Sie verstehen, warum die Bits von rechts nach links von 0 bis 7 durchnummeriert sind. Bit 0 ist „b_0" und entspricht 2^0. Bit 1 ist „b_1" und entspricht 2^1 usw.

Dezimal	Dual	Dezimal	Dual
0	00000000	32	00100000
1	00000001	33	00100001
2	00000010	•	
3	00000011	•	
4	00000100	•	
5	00000101	63	00111111
6	00000110	64	01000000
7	00000111	65	01000001
8	00001000	•	
9	00001001	•	
10	00001010	127	01111111
11	00001011	128	10000000
12	00001100	129	10000001
13	00001101		
14	00001110	•	
15	00001111		
16	00010000	•	
17	00010001	•	
•			
•			
•		254	11111110
31	00011111	255	11111111

Abb. 1.2: Dezimal-Dual Tabelle

Die dualen Äquivalente der Dezimalzahlen 0 bis 255 sind in Abb. 1–2 aufgelistet.

Aufgabe 1.1: Welchen Wert hat „11111100" in dezimaler Darstellung?

Dezimal nach Dual

Umgekehrt wollen wir jetzt das duale Äquivalent von dezimal „11" berechnen:

11 : 2 = 5 Rest 1 ⧫ 1 (niederwertigstes Bit)
5 : 2 = 2 Rest 1 ⧫ 1
2 : 2 = 1 Rest 0 ⧫ 0
1 : 2 = 0 Rest 1 ⧫ 1 (höchstwertiges Bit)

Das duale Äquivalent ist 1011 (dazu wird die Spalte ganz rechts von unten nach oben gelesen).

Das duale Äquivalent einer Dezimalzahl ergibt sich, indem man wiederholt durch 2 dividiert, bis man den Quotienten 0 erhält.

Aufgabe 1.2: Was ist 257 dual?
Aufgabe 1.3: Wandle 19 nach dual, dann zurück nach dezimal.

Rechnen mit Dualzahlen

Die Rechenregeln für Dualzahlen sind einfach. Die Regeln für die Addition sind:

$$\begin{aligned} 0 + 0 &= 0 \\ 0 + 1 &= 1 \\ 1 + 0 &= 1 \\ 1 + 1 &= (1)\,0 \end{aligned}$$

Hierbei bedeutet (1) einen Übertrag (carry) von 1 (beachten Sie, daß „10" das duale Äquivalent von dezimal „2" ist). Duale Subtraktion führt man aus durch „Addition des Komplements". Sie wird später erklärt, wenn wir die Darstellung negativer Zahlen lernen.

Beispiel:

$$\begin{array}{rr} (2) & 10 \\ +\,(1) & +\,01 \\ \hline (3) & 11 \end{array}$$

Wie im Dezimalsystem addiert man auch Dualzahlen, indem man rechts beginnend spaltenweise addiert.

Addition der rechten Spalte:

$$\begin{array}{r} 10 \\ +\,01 \\ \hline (0 + 1 = 1, \text{ kein Übertrag}) \end{array}$$

Addition der nächsten Spalte:

$$\begin{array}{r} 10 \\ +\,01 \\ \hline 11\ (1 + 0 = 1, \text{ kein Übertrag}) \end{array}$$

Aufgabe 1.4: Berechne 5 + 10 dual. Überprüfe, daß das Ergebnis 15 ist.

Einige weitere Beispiele dualer Addition:

$$\begin{array}{rlrl} 0010 & (2) & 0011 & (3) \\ +\,0001 & (1) & +\,0001 & (1) \\ \hline = 0011 & (3) & = 0100 & (4) \end{array}$$

Das letzte Beispiel veranschaulicht die Rolle des Übertrags.

Wenn wir die Bits ganz rechts betrachten:

$$1 + 1 = (1)\,0$$

Es tritt ein Übertrag von 1 auf, der zu den nächsten Bits addiert werden muß:

```
  001   – Spalte 0 wurde schon addiert
+ 000   –
+   1   – Übertrag
-------
=(1)0   – (1) zeigt einen neuen Übertrag in Spalte 2 an.
```

Das Endergebnis: 0100

Ein anderes Beispiel:

```
  0111   (7)
+ 0011   (3)
= 1010  (10)
```

In diesem Beispiel tritt wieder ein Übertrag auf, bis hin zur Spalte ganz links.

Aufgabe 1.5: Berechne das Ergebnis von

```
  1111
+ 0001
------
=    ?
```

Paßt das Ergebnis in vier Bit?

Mit acht Bit ist es demnach möglich, die Zahlen „00000000" bis „11111111", d. h. „0" bis „255", direkt darzustellen. Zwei Einschränkungen sollten wir uns dabei sofort verdeutlichen. Erstens stellen wir nur positive Zahlen dar, und zweitens ist die Größe der Zahlen auf 255 bgrenzt, wenn wir nur acht Bit verwenden. Wir wollen jetzt diese beiden Probleme nacheinander ansprechen.

Vorzeichenbehaftete Dualdarstellung

In der vorzeichenbehafteten Dualdarstellung verwendet man das Bit ganz links, um das Vorzeichen der Zahl zu kennzeichnen. Üblicherweise kennzeichnet man eine *positive* Zahl mit „0", eine *negative* mit „1". Jetzt bedeutet „11111111" −127, während „01111111" +127 bedeutet. Wir können jetzt positive und negative Zahlen darstellen, der Betrag der Zahlen ist aber auf maximal 127 begrenzt.

Beispiel: „0000 0001" bedeutet +1 (die führende „0" ist „+", gefolgt von „000 0001" = 1). „1000 0001" ist −1 (die führende „1" ist „−").

Aufgabe 1.6: Was ist −5 in vorzeichenbehafteter Dualdarstellung?

Jetzt wollen wir über das Problem der Größe sprechen: Um größere Zahlen darzustellen, wird es nötig sein, eine größere Anzahl Bits zu verwenden. Wenn wir zum Beispiel sechzehn Bit (zwei Byte) verwenden, können wir Zahlen von −32k bis +32k in dualer Form mit Vorzeichen darstellen (im Computerjargon bedeutet 1k 1024). Bit 15 wird für das Vorzeichen verwendet und die restlichen 15 Bit (Bit 14 bis Bit 0) für den

Betrag der Zahl: 2^{15} = 32k. Ist dieser Betrag immer noch zu klein, wird man drei oder mehr Byte verwenden. Wenn wir große ganze Zahlen darstellen wollen, dann ist es nötig, intern eine größere Zahl von Bytes zu verwenden. Deshalb stellen die meisten einfachen BASIC-Versionen und andere Sprachen für ganze Zahlen nur eine beschränkte Genauigkeit zur Verfügung. So können sie für die Zahlen, mit denen sie arbeiten, ein kürzeres internes Format verwenden. Bessere Versionen von BASIC oder von anderen Sprachen rechnen mit zusätzlichen signifikanten Dezimalstellen auf Kosten einer größeren Anzahl Bytes für jede Zahl.

Jetzt wollen wir uns um ein anderes Problem kümmern, nämlich das der möglichst effektiven Verarbeitungsgeschwindigkeit. Wir versuchen, eine Addition in der vorzeichenbehafteten Dualdarstellung, die wir dargestellt haben, auszuführen. Wir wollen „−5" und „+7" addieren.

+7 wird dargestellt durch	00000111
−5 wird dargestellt durch	10000101
Die duale Summe ist:	10001100 oder −12.

Dies ist jedoch nicht das richtige Ergebnis. Das richtige Ergebnis wäre +2. Um diese Darstellung richtig zu gebrauchen, muß man spezielle Maßnahmen ergreifen, die vom Vorzeichen abhängen. Dies führt zu größeren Schwierigkeiten und zu geringerer Effektivität. Mit anderen Worten: Die duale Addition vorzeichenbehafteter Zahlen funktioniert nicht richtig. Dies ist natürlich sehr ärgerlich, denn der Computer soll Information ja nicht nur darstellen, sondern auch damit rechnen.

Die Lösung dieses Problems ist die Darstellung als Zweierkomplement, die man statt der vorzeichenbehafteten Dualdarstellung verwendet. Vor Einführung des *Zweierkomplements* wollen wir erst einen Zwischenschritt betrachen, das *Einerkomplement.*

Einerkomplement

In der Darstellung als Einerkomplement erscheinen alle positiven ganzen Zahlen in ihrer korrekten dualen Form. Beispielsweise wird „+3" wie üblich als 00000011 dargestellt. Das Komplement „−3" erhält man jedoch dadurch, daß man jedes einzelne Bit invertiert. Jede 0 wird in eine 1 und jede 1 in eine 0 geändert. In unserem Beispiel ist „−3" dargestellt als Einerkomplement 11111100.

Ein anderes Beispiel:

+2 ist 00000010
−2 ist 11111101

Beachten Sie, daß positive Zahlen in dieser Darstellung links mit einer „0", negative links mit einer „1" beginnen.

Aufgabe 1.7: Die Darstellung von „+6" ist „00000110". Was ist die Darstellung von „−6" als Einerkomplement?

Als Test wollen wir minus 4 und plus 6 addieren:

−4 ist 11111011
+6 ist 00000110

Die Summe ist: (1) 00000001, wobei (1) einen Übertrag anzeigt.
Das richtige Ergebnis wäre „2" oder „00000010".
Ein weiterer Versuch:

−3 ist 11111100
−2 ist 11111101

Die Summe ist: (1)11111001
oder „1" und ein Übertrag. Das richtige Ergebnis wäre „−5". Die Darstellung von „−5" ist 11111010. Es funktionierte also nicht.
In der beschriebenen Form sind positive und negative Zahlen darstellbar. Das Ergebis einer normalen Addition kommt jedoch nicht immer „richtig" heraus. Deshalb werden wir noch eine andere Darstellung verwenden. Sie entsteht aus dem Einerkomplement und wird Zweierkomplement genannt.

Zweierkomplement

Beim Zweierkomplement werden positive Zahlen immer noch wie üblich vorzeichenbehaftet dual dargestellt, genau wie beim Einerkomplement. Der Unterschied liegt in der Darstellung *negativer Zahlen*. Eine negative Zahl im Zweierkomplement erhält man, indem man zuerst das Einerkomplement berechnet und dann *eins addiert*. Wir wollen dies an einem Beispiel illustrieren:
+3 ist in vorzeichenbehafteter Dualdarstellung 00000011. Das Einerkomplement davon ist 11111100. Durch Addition von Eins erhält man das Zweierkomplement 11111101.
Wir wollen eine Addition ausprobieren:

```
  (3)    00000011
+(5)   +00000101
-------------------
 =(8)   =00001000
```

Das Ergebnis stimmt.
Jetzt wollen wir die Subtraktion ausprobieren:

```
  (3)    00000011
(−5)   +11111011
-------------------
        =11111110
```

Wir prüfen das Ergebnis, indem wir das Zweierkomplement berechnen:
Das Einerkomplement von 11111110 ist 00000001
Addition von 1 + 1

Das Zweierkomplement ist also 00000010 oder +2.
Unser Ergebnis oben, „11111110" stellt „−2" dar. Es ist richtig!

Wir haben jetzt Addition und Subtraktion ausprobiert und die Ergebnisse waren richtig (wobei wir den Übertrag nicht beachtet haben). Es scheint, als ob das Zweierkomplement funktioniert!

Aufgabe 1.8: Was ist die Zweierkomplement-Darstellung von 127?

Aufgabe 1.9: Was ist die Zweierkomplement-Darstellung von −128?

Wir wollen jetzt +4 und −3 addieren (die Subtraktion wird ausgeführt, indem man das Zweierkomplement addiert):

```
+4 ist 00000100
−3 ist 11111101
-----------------
```

Das Ergebnis ist: (1) 00000001

Wenn wir den Übertrag nicht beachten, ist das Ergebnis 00000001, d. h. dezimal „1". Dies ist das richtige Ergebnis. Ohne einen vollständigen mathematischen Beweis zu geben, wollen wir einfach feststellen, daß diese Darstellung funktioniert. In der Darstellung als Zweierkomplement ist es möglich, vorzeichenbehaftete Zahlen zu addieren oder zu subtrahieren, ohne das Vorzeichen zu beachten. Wenn man die gebräuchlichen Regeln der dualen Addition anwendet, kommt das Ergebnis einschließlich des Vorzeichens richtig heraus. Ein Übertrag wird nicht beachtet. Dies ist ein wesentlicher Vorteil. Wäre das nicht der Fall, müßte man das Vorzeichen des Ergebnisses immer korrigieren, was Addition und Subtraktion wesentlich verlangsamen würde.

Der Vollständigkeit halber wollen wir feststellen, daß das Zweierkomplement die gebäuchlichste Darstellung für einfachere Prozessoren wie Mikroprozessoren ist. Für komplexe Prozessoren mag man andere Darstellungen verwenden. Man kann z. B. das Einerkomplement verwenden, benötigt aber spezielle Schaltungen, um das Ergebnis zu „berichtigen".

Von jetzt an werden alle vorzeichenbehafteten ganzen Zahlen stillschweigend als Zweierkomplemente dargestellt werden. Abb. 1.3 enthält eine Tabelle von Zweierkomplementen.

Aufgabe 1.10: Was sind die kleinste und die größte Zahl, die man als Zweierkomplement darstellen kann, wenn man nur ein Byte verwendet?

Aufgabe 1.11: Berechne das Zweierkomplement von 20. Berechne dann das Zweierkomplement des Ergebnisses. Ergibt sich wieder 20?

Das folgende Beispiel dient dazu, die Regeln für das Zweierkomplement verständlich zu machen. Insbesondere kennzeichnet C einen möglichen Übertrag. (C ist Bit 8 des Ergebnisses.)

V kennzeichnet einen Überlauf des Zweierkomplements, z. B. wenn das Vorzeichen „versehentlich" verändert wird, weil die Zahlen zu groß sind. Es ist ein wichtiger interner Übertrag von Bit 6 nach Bit 7 (dem Vorzeichenbit). Dies wird weiter unten erklärt.

+	2er-Komplement Kode	−	2er-Komplement Kode
+ 127	01111111	− 128	10000000
+ 126	01111110	− 127	10000001
+ 125	01111101	− 126	10000010
. . .		− 125	10000011
		. . .	
+ 65	01000001	− 65	10111111
+ 64	01000000	− 64	11000000
+ 63	00111111	− 63	11000001
. . .		. . .	
+ 33	00100001	− 33	11011111
+ 32	00100000	− 32	11100000
+ 31	00011111	− 31	11100001
. . .		. . .	
+ 17	00010001	− 17	11101111
+ 16	00010000	− 16	11110000
+ 15	00001111	− 15	11110001
+ 14	00001110	− 14	11110010
+ 13	00001101	− 13	11110011
+ 12	00001100	− 12	11110100
+ 11	00001011	− 11	11110101
+ 10	00001010	− 10	11110110
+ 9	00001001	− 9	11110111
+ 8	00001000	− 8	11111000
+ 7	00000111	− 7	11111001
+ 6	00000110	− 6	11111010
+ 5	00000101	− 5	11111011
+ 4	00000100	− 4	11111100
+ 3	00000011	− 3	11111101
+ 2	00000010	− 2	11111110
+ 1	00000001	− 1	11111111
+ 0	00000000		

Abb. 1.3: Tabelle der Zweierkomplemente

Wir wollen jetzt die Rolle des Übertrags „C" und des Überlaufs „V" demonstrieren.

Der Übertrag C

Hier ist ein Beispiel für einen Übertrag:

```
  (128)        10000000
+(129)       +10000001
-------------------------
 (257) = (1)  00000001
```

wobei (1) einen Übertrag anzeigt.

Das Ergebnis erfordert ein neuntes Bit (Bit „8", denn das Bit ganz rechts ist Bit „0"). Es ist das Übertragsbit.

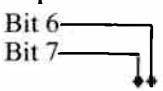
Wenn wir annehmen, daß das Übertragsbit das neunte Bit des Ergebnisses ist, erkennen wir das Ergebnis als 100000001 = 257.

Allerdings muß man das Übertragsbit mit Sorgfalt identifizieren und behandeln. Intern im Mikroprozessor sind die Register, in denen das Ergebnis gespeichert wird, allgemein nur acht Bit lang. Wenn das Ergebnis gespeichert wird, bleiben nur die Bits 0 bis 7 erhalten.

Deshalb bedarf ein Übertrag immer einer besonderen Behandlung: er muß mit speziellen Befehlen erkannt und dann bearbeitet werden. Bearbeitung des Übertrags heißt entweder, ihn irgendwo zu speichern (mit einem speziellen Befehl), ihn nicht zu beachten oder zu entscheiden, daß ein Fehler aufgetreten ist (wenn das größte zulässige Ergebnis „11111111" ist).

Überlauf V

Hier ist ein Beispiel für einen Überlauf:




```
  01000000     (64)
 +01000001    +(65)
-------------------------
 =10000001    =(-127)
```

Es wurde ein interner Übertrag von Bit 6 nach Bit 7 erzeugt. Dies nennt man einen Überlauf.

Das Ergebnis ist jetzt „versehentlich" negativ. Diese Situation muß erkannt werden, so daß man sie korrigieren kann.

Wir wollen einen anderen Fall überprüfen:

```
  11111111    (-1)
 +11111111   +(-1)
-------------------------
=(1)  11111110  =(-2)
 ▼
Übertrag
```

In diesem Fall trat ein interner Übertrag von Bit 6 nach Bit 7 auf und ebenso von Bit 7 nach Bit 8 (der formelle Übertrag C, den wir im vorhergehenden Abschnitt untersucht haben). Die Rechenregeln für Zweierkomplemente sagen, daß dieser Übertrag nicht beachtet werden soll. Dann ist das Ergebnis richtig.

Dies ist der Fall, weil der Übertrag von Bit 6 nach Bit 7 das Vorzeichenbit nicht verändert hat.

Dies ist dann kein *Überlauf*. Wenn man mit negativen Zahlen rechnet, ist ein Überlauf nicht ein einfacher Übertrag von Bit 6 nach Bit 7. Wir wollen ein weiteres Beispiel überprüfen:

```
        11000000    (−64)
       +10111111    (−65)
=(1)   01111111    (+127)
  ↓
Übertrag
```

In diesem Fall gab es keinen internen Übertrag von Bit 6 nach Bit 7, aber es trat ein externer Übertrag auf. Das Ergebnis ist falsch, weil sich Bit 7 geändert hat. Es sollte ein Überlauf angezeigt werden.

Ein Überlauf wird in vier Fällen auftreten:

1 – bei der Addition großer positiver Zahlen,
2 – bei der Addition großer negativer Zahlen,
3 – bei der Subtraktion einer großen positiven Zahl von einer großen negativen Zahl,
4 – bei der Subtraktion einer großen negativen Zahl von einer großen positiven Zahl.

Wir wollen jetzt unsere Definition des Überlaufs verbessern:

Der Überlaufanzeiger, ein spezielles Bit, das für diesen Zweck reserviert ist und „Flag" genannt wird, wird gesetzt, wenn ein Übertrag von Bit 6 nach Bit 7 auftritt und kein externer Übertrag vorliegt oder wenn kein Übertrag von Bit 6 nach Bit 7 auftritt, aber ein externer Übertrag vorliegt. Dies zeigt an, daß Bit 7, d. h. das Vorzeichenbit des Ergebnisses, fehlerhaft geändert wurde. Für den technisch erfahrenen Leser: Das Überlauf-Flag wird gesetzt durch eine Exklusiv-Oder-Verknüpfung des Übertragseingangs und des Übertragsausgangs von Bit 7 (dem Vorzeichenbit). Praktisch jeder Mikroprozessor verfügt über ein spezielles Überlauf-Flag, um diesen Fall anzuzeigen, der berichtigt werden muß.

Ein Überlauf zeigt an, daß das Ergebnis einer Addition oder Subtraktion mehr Bits zur Darstellung belegt, als in dem üblichen Acht-Bit-Register zur Verfügung stehen.

Der Übertrag und der Überlauf

Das Übertrags- und das Überlaufbit werden „Flags" genannt. Sie stehen in jedem Mikroprozessor zur Verfügung, und im nächsten Kapitel werden wir lernen, wie man sie zur effektiven Programmierung benutzt.

Diese beiden Anzeiger liegen in einem speziellen Register, das man Flag- oder „Statusregister" nennt. Dieses Register enthält noch weitere Anzeiger, deren Funktion im Kapitel 4 erklärt wird.

Beispiele

Wir wollen jetzt das Arbeiten von Übertrags- und Überlaufbit in konkreten Beispielen veranschaulichen. In jedem Beispiel kennzeichnet V den Überlauf und C den Übertrag.

Trat kein Überlauf auf, ist V = 0. Trat ein Überlauf auf, ist V = 1 (entsprechend auch der Übertrag C). Beachten Sie, daß der Übertrag nach den Regeln für das Zweierkomplement nicht beachtet werden soll. (Ein mathematischer Beweis wird hier nicht angegeben.)

Positiv-positiv

```
  00000110   (+6)
+ 00001000   (+8)
-------------------------------
= 00001110   (+14)  V:0  C:0
```

(richtig)

Positiv-positiv mit Überlauf

```
  01111111   (+127)
+ 00000001   (+1)
-------------------------------
= 10000000   (-128)  V:1  C:0
```

Obiges Ergebnis ist falsch, da ein Überlauf auftrat.

(falsch)

Positiv-negativ (Ergebnis positiv)

```
     0000100   (+4)
+    1111110   (-2)
-------------------------------
=(1) 0000010   (+2)  V:0  C:1 (nicht beachten)
```

(richtig)

Positiv-negativ (Ergebnis negativ)

```
  00000010   (+2)
+ 11111100   (-4)
-------------------------------
= 11111110   (-2)  V:0  C:0
```

(richtig)

Negativ-negativ

```
     11111110   (-2)
+    11111100   (-4)
-------------------------------
=(1) 11111010   (-6)  V:0  C:1 (nicht beachten)
```

(richtig)

Negativ-negativ mit Überlauf

```
      10000001   (−127)
+     11000010   (−62)
-----------------------------------------
=(1) 01000011    (+67)   V:1   C:1
```

(falsch)

Hier trat ein „Unterlauf" auf, da zwei große negative Zahlen addiert wurden. Das richtige Ergebnis wäre −189, was zur Darstellung in acht Bit zu groß ist.

Aufgbe 1.12: Vervollständige die folgenden Additionen. Gib das Ergebnis an, den Übertrag C, den Überlauf V und ob das Ergebnis richtig ist.

```
  10111111  (     )
 +11000001  (     )
-----------------------
=           V:    C:
☐ richtig  ☐ falsch
```

```
  11111010  (     )
 +11111001  (     )
-----------------------
=           V:    C:
☐ richtig  ☐ falsch
```

```
  00010000  (     )
 +01000000  (     )
-----------------------
=           V:    C:
☐ richtig  ☐ falsch
```

```
  01111110  (     )
 +00101010  (     )
-----------------------
=           V:    C:
☐ richtig  ☐ falsch
```

Aufgabe 1.13: Kann man ein Beispiel angeben, bei dem bei der Addition einer positiven und einer negativen Zahl ein Überlauf auftritt? Warum?

Festkomma-Darstellung

Jetzt wissen wir, wie man vorzeichenbehaftete ganze Zahlen darstellt. Das Problem der Größe haben wir jedoch noch nicht gelöst. Wollen wir größere ganze Zahlen darstellen, benötigen wir mehrere Bytes. Um arithmetische Operationen effektiv auszuführen, ist es notwendig, eine feste Zahl von Bytes zu verwenden, keine variable. Wenn aber die Zahl der Bytes einmal festgelegt ist, ist auch die maximale Größe der Zahlen, die man darstellen kann, festgelegt.

Aufgabe 1.14: Was sind die größte und die kleinste Zahl, die man in zwei Byte mit Zweierkomplementen darstellen kann?

Das Problem der Größe

Bei der Addition von Zahlen haben wir uns auf acht Bit beschränkt, weil der Prozessor, den wir verwenden wollen, intern jeweils acht Bit gleichzeitig verarbeitet. Dies beschränkt uns jedoch auf Zahlen im Bereich von −128 bis +127. Für viele Anwendungen reicht das natürlich nicht aus. Mehrfache Genauigkeit wird verwendet, um die Stellenzahl, die dargestellt werden kann, zu erhöhen. Dann kann ein Zwei-, Drei- oder

N-Byte-Format verwendet werden. Wir wollen als Beispiel ein „doppeltgenaues“ 16-Bit Format untersuchen:

00000000	00000000	ist „0“
00000000	00000001	ist „1“
. . .		
01111111	11111111	ist „32767“
11111111	11111111	ist „−1“
11111111	11111110	ist „−2“

Aufgabe 1.15: Was ist die größte negative ganze Zahl, die in dreifachgenauem Format als Zweierkomplement dargestellt werden kann?

Diese Methode bringt jedoch auch Nachteile mit sich. Wenn man beispielsweise zwei Zahlen addiert, muß man sie generell in Stücken zu acht Bit addieren. Dies wird in Kapitel 3 (Grundlegende Techniken der Programmierung) erklärt werden. Das führt zu langsamerer Verarbeitung. Außerdem belegt diese Darstellung für jede Zahl 16 Bit, selbst wenn sie mit nur acht Bit dargestellt werden könnte. Es ist deshalb üblich, 16 oder vielleicht 32 Bit zu verwenden, aber selten mehr.

Wir wollen den folgenden wichtigen Punkt bedenken: Welche Zahl von Bit n für die Darstellung als Zweierkomplement auch immer gewählt wird, sie ist dann festgelegt. Wenn als Ergebnis oder bei irgendeiner Zwischenrechnung eine Zahl herauskommt, die länger als n Bit ist, gehen einige Bit verloren. Das Programm rettet normalerweise die n Bit links (die wichtigsten) und läßt die Bits von niedrigerer Ordnung weg. Dies nennt man Abschneiden des Ergebnisses.

Hier ist ein Beispiel im Dezimalsystem, bei dem eine sechsstellige Darstellung verwendet wird:

```
   12345 6
×      1,2
-----------
   24691 2
  123456
-----------
=148147,2
```

Das Ergebnis benötigt sieben Stellen! Die „2“ nach dem Komma fällt heraus und das Endergebnis ist 148147. Es wurde abgeschnitten. Solange die Position des Kommas nicht verloren geht, wendet man üblicherweise diese Methode an, um den Zahlenbereich für die Rechnungen auf Kosten der Genauigkeit zu erweitern.

Im Dualsystem ist das Problem das gleiche. Die Einzelheiten der dualen Multiplikation werden in Kapitel 4 gezeigt.

Die Darstellung in festem Format kann einen Genauigkeitsverlust verursachen, aber sie wird für übliche Berechnungen und mathematische Operationen ausreichen.

Leider ist bei der Buchhaltung kein Verlust an Genauigkeit tragbar. Wenn zum Beispiel die Registrierkasse einem Kunden eine große Sum-

me anzeigt, ist es nicht akzeptabel, wenn er eine fünfstellige Summe bezahlen muß, die auf eine Mark gerundet ist. Wo immer die Genauigkeit des Ergebnisses wesentlich ist, muß man eine andere Darstellung verwenden. Die Lösung, die normalerweise verwendet wird, heißt BCD oder binär kodierte Dezimaldarstellung.

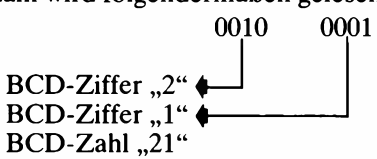
BCD-Darstellung

Das Prinzip der Zahlendarstellung in BCD ist es, jede Dezimalziffer getrennt zu kodieren und so viele Bits zu verwenden, wie zur genauen Darstellung der vollständigen Zahl nötig sind. Um die Ziffern 0 bis 9 zu kodieren, braucht man vier Bit. Mit drei Bit ergeben sich nur acht Kombinationen, man kann damit keine zehn Ziffern kodieren. Vier Bit ergeben 16 Kombinationen und reichen zur Kodierung der Ziffern „0" bis „9" aus. Festzuhalten ist auch, daß sechs der möglichen Kodes in der BCD-Darstellung nicht verwendet werden (siehe Abb. 1.4). Dies führt später beim Addieren und Subtrahieren zu einem Problem, das wir lösen müssen. Weil nur vier Bit benötigt werden, um eine BCD-Ziffer zu kodieren, kann man in jedem Byte zwei BCD-Ziffern unterbringen. Dies nennt man „gepackte" BCD-Darstellung.

KODE	BCD SYMBOL	KODE	BCD SYMBOL
0000	0	1000	8
0001	1	1001	9
0010	2	1010	unused
0011	3	1011	unused
0100	4	1100	unused
0101	5	1101	unused
0110	6	1110	unused
0111	7	1111	unused

Abb. 1.4: BCD Tabelle

Beispielsweise bedeutet „00000000" in BCD „00". „10011001" ist „99". Eine BCD-Zahl wird folgendermaßen gelesen:

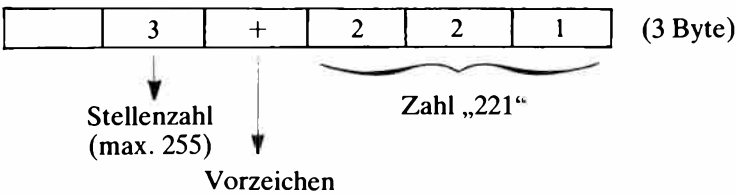


Aufgabe 1.16: Was ist die BCD-Darstellung von „29", von „91"?

Aufgabe 1.17: Ist „10100000" eine zulässige BCD-Darstellung? Warum?

Es werden so viele Bytes verwendet, wie zur Darstellung aller BCD-Ziffern nötig sind. Typischerweise verwendet man ein oder mehrere Nibble am Anfang der Darstellung, um die Gesamtzahl der Nibble, d. h. der verwendeten BCD-Ziffern anzugeben. Ein weiteres Nibble oder Byte wird verwendet, um die Position des Kommas anzugeben. Das Format kann jedoch auf verschiedene Art festgelegt werden.

Hier ist ein Beispiel für eine BCD-Darstellung ganzer Zahlen aus mehreren Bytes:



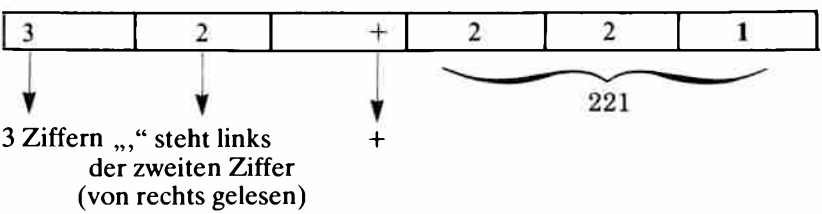
Dies stellt +221 dar.

Das Vorzeichen kann z. B. durch 0000 für + und 0001 für − festgelegt werden.

Aufgabe 1.18: Stelle „−23123" in der gleichen Konvention dar. Verwende zuerst das BCD-Format wie oben, dann die binäre Form.

Aufgabe 1.19: Stelle „222" und „111" sowie das Ergebnis von „222x111" im BCD-Format dar. (Berechne das Ergebnis von Hand und wandle es in die obige Darstellung um.)

Die BCD-Darstellung kann leicht um Dezimalbrüche erweitert werden. Beispielsweise kann man +2.21 darstellen als



Der Vorteil von BCD ist, daß man absolut genaue Ergebnisse erhält. Sein Nachteil ist, daß viel Speicher benötigt wird, und die Rechenoperationen langsam sind. Dies ist für kaumännische Anwendungen akzeptabel, wird für andere Anwendungen jedoch normalerweise nicht verwendet.

Aufgabe 1.20: Wieviele Bit benötigt man, um „9999" in BCD-Form darzustellen, wieviele in der Darstellung als Zweierkomplement?

Wir haben jetzt die Probleme gelöst, die mit der Darstellung von ganzen Zahlen, von vorzeichenbehafteten ganzen Zahlen und auch von großen ganzen Zahlen zusammenhängen. Wir haben auch schon eine Methode vorgestellt, Dezimalbrüche in BCD-Form darzustellen. Jetzt wollen wir das Problem angehen, wie Dezimalbrüche in einem Format fester Länge dargestellt werden können.

Gleitkomma-Darstellung

Das Grundprinzip ist es, Dezimalbrüche in einem festen Format darzustellen. Um keine Bits zu verschwenden, werden alle Zahlen normalisiert.

„0,000123" verbraucht z. B. auf der linken Seite drei Stellen für Nullen, die keine andere Funktion haben, als die Position des Kommas anzugeben. Die Normalisierung führt zu $0{,}123 \times 10^{-3}$. „0,123" wird *normalisierte Mantisse* genannt, „−3" ist der *Exponent*. Wir haben diese Zahl normalisiert, indem wir alle überflüssigen Nullen auf der linken Seite entfernt und den Exponenten entsprechend angepaßt haben.

Wir wollen ein anderes Beispiel betrachten:

22,1 normalisiert ergibt $0{,}221 \times 10^2$

oder $M \times 10^E$, wobei M die Mantisse und E der Exponent ist.

Man sieht sofort, daß eine normalisierte Dezimalzahl immer durch eine Mantisse kleiner 1 und größer oder gleich 0,1 charakterisiert wird, wenn die Zahl von Null verschieden ist. Mathematisch kann man dies darstellen durch

$$0{,}1 \leq M < 1 \text{ oder } 10^{-1} < M < 10^0.$$

In binärer Darstellung ergibt sich entsprechend

$$2^{-1} \leq M < 2^0 \text{ (oder } 0{,}5 \leq M < 1).$$

Dabei ist M der Betrag der Mantisse (d. h. ohne das Vorzeichen).

Beispiel: 111,01 normalisiert ergibt $0{,}11101 \times 2^3$.

Die Mantisse ist 0,11101, der Exponent ist 3.

Nachdem wir das Prinzip der Gleitkommadarstellung festgelegt haben, wollen wir uns jetzt die tatsächliche Darstellung anschauen. Eine typische Gleitkommadarstellung ist unten gezeigt.

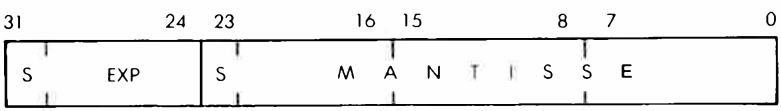


Abb. 1.5: Typische Gleitkommadarstellung

In diesem Beispiel werden insgesamt vier Byte, d. h. 32 Bit benutzt. Das erste Byte links wird verwendet, um den Exponenten darzustellen. Sowohl der Exponent als auch die Mantisse werden als Zweierkomplement

dargestellt. Daraus ergibt sich ein maximaler Exponent von 127. „S" in Abb. 1.5 kennzeichnet das Vorzeichenbit.

Zur Darstellung der Mantisse werden drei Byte verwendet. Da das erste Bit des Zweierkomplements das Vorzeichen angibt, bleiben 23 Bit, um den Betrag der Mantisse anzugeben.

Aufgabe 1.21: Wie viele Dezimalstellen kann man in einer Mantisse von 23 Bit darstellen?

Dies ist nur ein Beispiel für eine Gleitkommadarstellung. Es ist auch möglich, nur drei Byte zu verwenden, oder es ist möglich, mehr zu verwenden. Die obige Darstellung mit vier Byte ist aber weit verbreitet und bedeutet einen brauchbaren Kompromiß aus Genauigkeit, Größe der Zahlen, Speicherbedarf und Rechengeschwindigkeit.

Wir haben jetzt die Probleme, die mit der Zahlendarstellung zusammenhängen, untersucht, und wir wissen, wie man ganze Zahlen, vorzeichenbehaftete Zahlen und Dezimalbrüche darstellt. Jetzt wollen wir überlegen, wie alphanumerische Daten intern dargestellt werden können.

Darstellung alphanumerischer Daten

Die Darstellung alphanumerischer Daten, d. h. von Zeichen ist ganz einfach: Alle Zeichen werden in einem acht-Bit-Kode kodiert. In der Computerwelt werden nur zwei Kodes allgemein verwendet, der ASCII-Kode und der EBCDIC-Kode. ASCII heißt „American Standard Code for Information Interchange" und wird bei den Mikroprozessoren universell verwendet. EBCDIC ist eine Variation von ASCII, die von IBM verwendet wird und deshalb für Mikroprozessoren nicht verwendet wird, es sei denn, man will ein IBM-Terminal anschließen.

Wir wollen kurz den ASCII-Kode untersuchen. Wir müssen 26 Großbuchstaben und ebensoviele Kleinbuchstaben des Alphabets kodieren, außerdem zehn Ziffern und etwa 20 weitere Sonderzeichen. Dies kann man leicht mit 7 Bit tun, die 128 mögliche Kodes zulassen (siehe Abb. 1.6). Deshalb werden alle Zeichen mit 7 Bit kodiert. Das achte Bit ist das *Paritätsbit,* wenn es überhaupt verwendet wird. Die Parität ist ein Verfahren, mit dem überprüft wird, daß der Inhalt eines Bytes nicht fälschlicherweise verändert wurde. Die Zahl der Einsen in dem Byte wird gezählt und das achte Bit auf eins gesetzt, wenn das Ergebnis ungerade war. Dadurch wird die Gesamtzahl gerade. Dies nennt man dann gerade Parität. Man kann auch ungerade Parität verwenden, d. h. das achte Bit so setzen, daß die Gesamtzahl von Einsen in dem Byte ungerade wird.

Beispiel: Wir wollen das Paritätsbit zu „0010011" für gerade Parität berechnen. Die Anzahl der Einsen ist drei. Das Paritätsbit muß also eine Eins sein, so daß sich die Gesamtzahl vier, d. h. eine gerade Zahl ergibt. Das Ergebnis ist 10010011, wobei die führende Eins das Paritätsbit ist und 0010011 das Zeichen charakterisiert.

Abb. 1.6 zeigt eine Tabelle der 7-Bit-ASCII-Kodes. Man verwendet sie in der Praxis entweder „wie sie ist" d. h. ohne Parität, indem man eine

Null links anfügt, oder aber mit Parität, indem man das entsprechende Bit links anfügt.

Aufgabe 1.22: Berechne die 8-Bit-Darstellung der Ziffern „0" bis „9" mit gerader Parität. (Dieser Kode wird in den Anwendungsbeispielen in Kapitel 8 verwendet.)

Aufgabe 1.23: Verfahre ebenso mit den Buchstaben „A" bis „F".

Aufgabe 1.24: Gib die Binärdarstellung der folgenden vier Zeichen an. Dabei soll der ASCII-Kode ohne Parität verwendet werden (d. h. das Bit ganz links ist „0").

„A"
„?"
„3"
„b"

HEX	MSD	0	1	2	3	4	5	6	7
LSD	BITS	000	001	010	011	100	101	110	111
0	0000	NUL	DLE	SPACE	0	@	P	–	p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	'	7	G	W	g	w
8	1000	BS	CAN	(	8	H	X	h	x
9	1001	HT	EM	)	9	I	Y	i	y
A	1010	LF	SUB	*	:	J	Z	j	z
B	1011	VT	ESC	+	;	K	[	k	{
C	1100	FF	FS	,	<	L	\	l	|
D	1101	CR	GS	–	=	M	]	m	}
E	1110	SO	RS	.	>	N	^	n	~
F	1111	SI	US	/	?	O	←	o	DEL

Abb. 1.6: ASCII-Umwandlungs-Tabelle

Für spezielle Probleme wie Datenübertragung kann man andere Kodes verwenden, z. B. fehlerkorrigierende Kodes. Dies liegt jedoch außerhalb des Rahmens dieses Buchs.

Wir haben jetzt die gebräuchlichen Darstellungen von Programmen und von Daten innerhalb des Computers untersucht. Jetzt wollen wir die mögliche externe Darstellung betrachten.

Externe Darstellung von Information

Die externe Darstellung bezieht sich auf die Art und Weise, wie Infor-

mation an den Benutzer, d. h. im allgemeinen an den Programmierer übergeben wird. Extern kann Information im wesentlichen in drei Formaten vorliegen: binär, oktal oder hexadezimal und symbolisch.

1. Binär

Es wurde gezeigt, daß Information intern in *Bytes* gespeichert wird, die aus Sequenzen von acht Bit (Nullen oder Einsen) bestehen. Manchmal ist es günstig, diese interne Information direkt in ihrem binären Format anzuzeigen, und dies nennt man dann *binäre Darstellung*. Ein einfaches Beispiel sind Leuchtdioden (LEDs), d. h. kleine Lämpchen an der Frontplatte des Computers. Bei einem Acht-Bit-Mikroprozessor sind auf der Frontplatte typischerweise acht LEDs untergebracht, die den Inhalt eines internen Registers anzeigen. (Ein Register wird dazu verwendet, acht Bit Information zu speichern. Es wird in Kapitel 2 näher beschrieben.) Eine LED, die leuchtet, zeigt eine Eins an, eine Null wird durch eine dunkle LED dargestellt. Eine solche Binärdarstellung mag für die detaillierte Fehlersuche in einem komplexen Programm vorteilhaft sein, speziell wenn es Eingaben und Ausgaben enthält, aber sie ist für den Benutzer natürlich unpraktisch. Deshalb betrachtet man in den meisten Fällen Information lieber in symbolischer Form. „9" ist viel leichter zu verstehen als „1001". Es wurden noch weitere Darstellungen entwickelt, die den Kontakt zwischen Mensch und Maschine erleichtern.

Oktal und Hexadezimal

„Oktale" und „hexadezimale" Darstellung fassen jeweils drei bzw. vier Bit in einem Symbol zusammen. Im Oktalsystem wird jede Kombination aus drei binären Bits durch eine Zahl zwischen 0 und 7 dargestellt.

„Oktal" ist ein Format, das drei Bit verwendet, wobei jede Kombination der drei Bit durch ein Symbol zwischen 0 und 7 charakterisiert wird:

dual	oktal
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Abb. 1.7: Oktale Symbole

Beispielsweise wird „00 100 100“ binär dargestellt als

0 4 4

oder „044“ oktal.

Ein anderes Beispiel: 11 111 111 ist

3 7 7

oder „377“ oktal.

Umgekehrt bedeutet oktal „211“:

010 001 001

oder „1001001“ binär.

Oktales Format wurde traditionell auf älteren Rechnern verwendet, die mit einer unterschiedlichen Zahl von Bits im Bereich zwischen acht und 64 rechneten. In jüngerer Zeit wurde das Acht-Bit-Format mit dem Übergewicht der Acht-Bit-Mikroprozessoren Standard, und man verwendet eine andere praktischere Darstellung, das *hexadezimale Format*.

In der hexadezimalen Darstellung wird eine Gruppe von vier Bit durch eine Hexadezimalziffer charakterisiert. Hexadezimalziffern stellt man mit den Ziffern 0 bis 9 und den Buchstaben A, B, C, D, E und F dar. Beispielsweise stellt man „0000“ durch „0“ dar, „0001“ durch „1“ und „1111“ durch den Buchstaben „F“ (siehe Abb. 1.8).

Beispiel: 1010 0001 binär wird durch

A 1 hexadezimal dargestellt.

Aufgabe 1.25: Was ist die Hexadezimaldarstellung von „10101010“?

Aufgabe 1.26: Was ist umgekehrt das binäre Aquivalent zu hexadezimal „FA“?

Aufgabe 1.27: Wie sieht die oktale Darstellung von „01000001“ aus?

Der Vorteil des hexadezimalen Formats ist es, daß acht Bit mit nur zwei Ziffern dargestellt werden. Dies kann man sich leichter vorstellen oder merken und schneller in den Computer eintippen als das binäre Äquivalent. Deshalb ist die hexadezimale Form bei den meisten neueren Mikrocomputern die bevorzugte Methode, um Gruppen von vier Bit darzustellen.

Hat die Information, die im Speicher steht, eine spezielle Bedeutung, z. B. Text oder Zahlen, ist die hexadezimale Form natürlich nicht angebracht, um den Sinn dieser Information darzustellen, wenn sie für den Benutzer ausgegeben wird.

Symbolische Darstellung

Symbolische Darstellung bezieht sich auf die externe Darstellung von Information in symbolischer Form. Beispielsweise stellt man Dezimalzah-

DEZIMAL	DUAL	HEX	OKTAL
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	8	10
9	1001	9	11
10	1010	A	12
11	1011	B	13
12	1100	C	14
13	1101	D	15
14	1110	E	16
15	1111	F	17

Abb. 1.8: Hexadezimalkode

len als Dezimalzahlen dar und nicht als Folge von hexadezimalen Symbolen oder von Bits. Ebenso wird Text als solcher ausgegeben. Natürlich ist die symbolische Darstellung für den Benutzer am bequemsten. Sie wird immer dann verwendet, wenn ein entsprechendes Ausgabegerät wie ein Bildschirm oder ein Drucker zur Verfügung steht. (Ein Bildschirm funktioniert ähnlich wie ein Fernsehgerät und wird zur Anzeige von Text oder von Grafik verwendet.) Für kleinere Systeme wie Einplatinencomputer ist es leider oft zu teuer, solche Anzeigen vorzusehen, und der Benutzer ist auf hexadezimale Kommunikation mit dem Computer beschränkt.

Zusammenfassung der externen Darstellungen

Die symbolische Darstellung von Information ist am günstigsten, weil sie für den Benutzer die natürlichste Form ist. Jedoch wird dazu ein teures Interface in Form einer alphanumerischen Tastatur und eines Drukkers oder eines Bildschirms benötigt. Aus diesem Grund ist sie bei den billigeren Systemen oft nicht vorhanden. Dann verwendet man eine andere Darstellung , und in diesem Fall herrscht das hexadezimale Format vor. Nur in seltenen Fällen, wenn es um die Fehlersuche in Hardware oder Software geht, wird die binäre Darstellung verwendet. Dabei wird der Inhalt von Registern oder Speichern direkt binär angezeigt.

(Die Frage, ob eine direkte binäre Anzeige auf der Frontplatte von Vorteil ist, war immer Gegenstand hitziger Debatten. Darauf soll hier aber nicht eingegangen werden.)

Wir haben jetzt gesehen, wie Information intern und extern dargestellt wird. Wir werden jetzt einen echten Mikroprozessor untersuchen, der diese Information verarbeitet.

Zusätzliche Aufgaben

Aufgabe 1.28: Was ist der Vorteil des Zweierkomplements zur Darstellung von vorzeichenbehafteten Zahlen gegenüber anderen Darstellungen?

Aufgabe 1.29: Wie würden Sie „1024" in direktem dualen Format darstellen, wie in vorzeichenbehaftetem dualen Format und wie als Zweierkomplement?

Aufgabe 1.30: Was ist das V-Bit? Sollte es der Programmierer nach einer Addition oder Subtraktion testen?

Aufgabe 1.31: Berechne das Zweierkomplement von „+16", „+17", „+18", „−16", „−17" und „−18".

Aufgabe 1.32: Gib die hexadezimale Darstellung des folgenden Textes an, der intern im ASCII-Format ohne Parität gespeichert ist: „NACHRICHT".

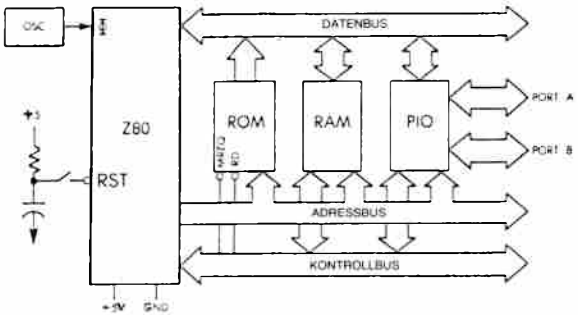


Abb. 2.1: Z80-Standardsystem

Die MPU hat drei *Busse*: einen 8 Bit breiten bidirektionalen *Datenbus,* der oben abgebildet ist, einen 16 Bit breiten unidirektionalen *Adressbus* und einen *Kontrollbus*, der im unteren Teil der Abbildung dargestellt ist. Wir wollen die Funktion jedes dieser Busse erklären.

Der *Datenbus* überträgt die Daten, die zwischen verschiedenen Elementen des Systems ausgetauscht werden. Typischerweise überträgt er Daten vom Speicher zur MPU, von der MPU zum Speicher oder von der MPU zu einem Ein-/Ausgabebaustein. (Ein Ein-/Ausgabebaustein hat die Aufgabe, mit einem externen Gerät zu kommunizieren.)

Der *Adreßbus* überträgt eine Adresse, die von der MPU erzeugt wird und die ein Register auswählt, das an das System angeschlossen ist. Die Adresse legt das Ziel oder die Quelle der Daten fest, die über den Datenbus übertragen werden.

Der *Kontrollbus* übermittelt die verschiedenen Signale, die zur Synchronisation des Systems gebraucht werden.

Nachdem wir den Sinn der Busse beschrieben haben, wollen wir jetzt die zusätzlichen Bausteine anschließen, die für das vollständige System nötig sind.

Jede MPU benötigt eine genaue zeitliche Referenz, die aus einem *Quarz* und einem *Taktgeber* besteht. In den meisten „älteren" Mikroprozessoren ist der Taktgeber außerhalb der MPU und besteht aus einem zusätzlichen Baustein. In den moderneren Mikroprozessoren ist der Taktgeber meistens in die MPU integriert. Der Quarzkristall befindet sich jedoch wegen seiner Größe immer außerhalb der MPU. Quarz und Taktgeber erscheinen in Abb. 2.1 links von der MPU.

Jetzt wollen wir unsere Aufmerksamkeit den anderen Elementen des Systems zuwenden. Wir gehen die Zeichnung von links nach rechts durch und erkennen:

Das *ROM* (read-only-memory) ist der *Lesespeicher* und enthält das Programm für das System. Der ROM-Speicher hat den Vorteil, daß sein Inhalt beständig ist und beim Ausschalten des Systems nicht verschwindet.

2

Z80 Hardware Organisation

Einführung

Um auf einem elementaren Niveau zu programmieren, braucht man die interne Struktur des Prozessors, den man verwendet, nicht im einzelnen zu verstehen. Um jedoch effizient zu programmieren, ist ein solches Verständnis nötig. Das Ziel dieses Kapitels ist es, das grundlegende Hardwarekonzept des Z80 vorzustellen, das man benötigt, um die Arbeitsweise des Z80-Systems zu verstehen. Ein vollständiges Mikrocomputersystem enthält nicht nur den Mikroprozessor (hier den Z80), sondern auch andere Bausteine. Dieses Kapitel präsentiert den eigentlichen Z80, während die anderen Bausteine (hauptsächlich Ein-/Ausgabebausteine) in einem anderen Kapitel (Kapitel 7) erklärt werden.

Wir werden hier die grundlegende Architektur des Mikroprozessorsystems besprechen und dann die interne Organisation des Z80 genauer studieren. Wir werden zum Teil die verschiedenen Register untersuchen. Wir werden dann den Mechanismus der Programmausführung und -kontrolle studieren. Vom Standpunkt der Hardware ist dieses Kapitel nur eine vereinfachte Darstellung. Der Leser, der ein tiefergehendes Verständnis erwerben will, sei auf unser Buch „Chip und System. Einführung in die Mikroprozessoren-Technik" (Ref.-Nr. 3017) verwiesen.

Der Z80 wurde entworfen, um den 8080 zu ersetzen und zusätzliche Eigenschaften anzubieten. In diesem Kapitel wird des öfteren auf den Entwurf des 8080 verwiesen werden.

Systemarchitektur

Die Architektur eines Mikrocomputersystems ist in Abb. 2.1 dargestellt. Der Mikroprozessor (MPU), hier ein Z80, ist auf der linken Seite abgebildet. Er realisiert die Funktionen einer Zentraleinheit (CPU) auf einem einzelnen Chip. Der Mikroprozessor enthält eine Arithmetik-Logik-Einheit (ALU) mit ihren internen Registern und die Steuereinheit (CU), die den Ablauf des Systems steuert. Ihr Arbeiten wird in diesem Kapitel erklärt.

Das ROM enthält deshalb immer einen *Urlader* (bootstrap) oder einen *Monitor* (dessen Funktion später erklärt wird), die den Start des Systems erlauben. In einer Anlage für Prozeßsteuerung werden nahezu alle Programme in ROMs gespeichert, da sie fast nie geändert werden. In einem solchen Fall muß der Benutzer das System von Fehlern in der Stromversorgung schützen. Das Programm darf nicht flüchtig sein. Es muß in einem ROM stehen.

Bei einem Hobby-Anwender jedoch oder bei der Programmentwicklung (wo der Programmierer sein Programm testet) stehen die meisten Programme im RAM, so daß sie leicht geändert werden können. Später können sie im RAM bleiben, oder in das ROM übernommen werden, wenn das gewünscht wird. Das RAM ist allerdings flüchtig. Sein Inhalt geht verloren, wenn die Spannung abgeschaltet wird.

Das *RAM* (random-access-memory) ist der *Schreib-/Lesepeicher* des Systems. Ein System zur Steuerung enthält normalerweise nur wenig RAM (nur für Daten). Ein System zur Programmentwicklung wird andererseits viel RAM enthalten, da es Programme und Software zur Unterstützung der Entwicklung enthält. Vor dem Arbeiten muß der gesamte Inhalt des RAM von einem externen Gerät geladen werden.

Schließlich enthält das System einen oder mehrere Interfacebausteine, so daß es mit der Umwelt in Verbindung treten kann. Der am häufigsten verwendete Interfacebaustein ist die PIO oder der *Parallel-Ein-/Ausgabe-Baustein*. In der Zeichnung ist eine PIO abgebildet. Wie alle anderen Bausteine des Systems ist die PIO mit allen drei Bussen verbunden und sie stellt wenigstens zwei 8-Bit-Ports zur Kommunikation mit der Außenwelt zur Verfügung. Weitergehende Einzelheiten über die Funktion einer PIO können dem Buch „Chip und System", speziell für das Z80 System dem Kapitel 7 (Ein-/Ausgabebausteine) entnommen werden.

Alle diese Bausteine sind mit allen drei Bussen einschließlich des Kontrollbusses verbunden. Um die Zeichnung nicht zu unübersichtlich zu machen, sind die Verbindungen zwischen Kontrollbus und den verschiedenen Bausteinen nicht eingezeichnet.

Die Funktionsbausteine, die wir beschrieben haben, brauchen nicht notwendigerweise aus einzelnen hochintegrierten Schaltkreisen zu bestehen. Wir können auch *Kombinationsbausteine* verwenden, die sowohl eine PIO als auch eine begrenzte Menge von ROM oder RAM enthalten.

Für ein reales System sind noch weitere Bausteine notwendig. Im einzelnen müssen die Busse normalerweise *gepuffert* werden. Für die Speicherbausteine mag eine *Dekodierlogik* verwendet werden, und schließlich werden eventuell bestimmte Signale mit *Treibern* verstärkt. Diese zusätzlichen Schaltungen werden hier nicht besprochen, weil sie für die Programmierung keine Bedeutung haben. Der Leser, der speziell am Aufbau und an der Interfacetechnik interessiert ist, sei auf das Buch „Mikroprozessor Interface Techniken" (Ref.-Nr. 3012) verwiesen.

Im Innern eines Mikroprozessors

Die große Mehrheit der Mikroprozessoren, die heute auf dem Markt ist, besitzt die gleiche Architektur. Diese „Standardarchitektur" wird hier beschrieben. Sie ist in Abb. 2.2 dargestellt. Die Module dieses Standardmikroprozessors werden jetzt von rechts nach links im einzelnen erklärt.

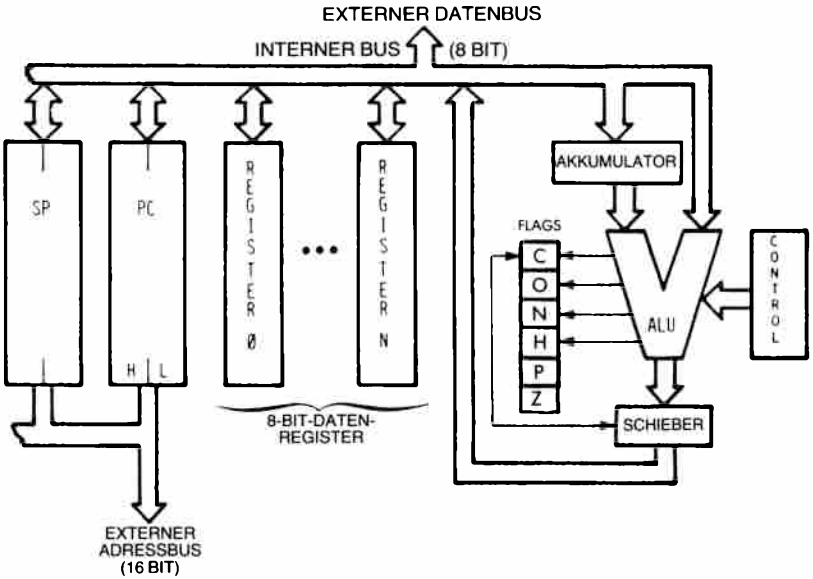


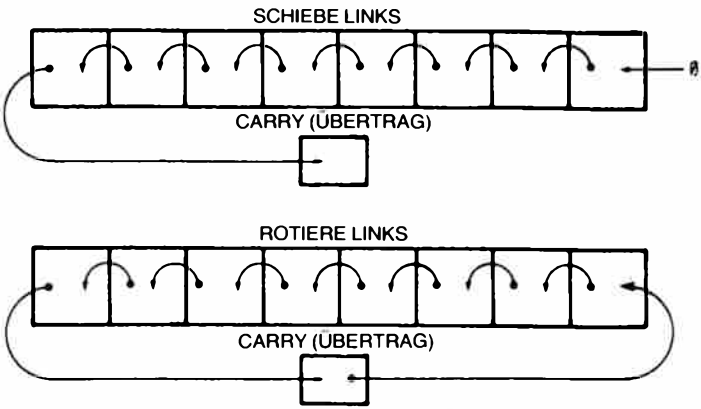
Abb. 2.2: „Standardarchitektur" eines Mikroprozessors

Das Kästchen *„Steuerung"* auf der rechten Seite stellt die Steuereinheit dar, die das gesamte System synchronisiert. Ihre Aufgabe wird innerhalb dieses Kapitels erklärt werden.

Die *ALU* führt arithmetische und logische Operationen durch. Ein spezielles Register versorgt einen der Eingänge der ALU, hier den linken Eingang. Er wird Akkumulator genannt. (Es können auch mehrere Akkumulatoren vorhanden sein.) Der Akkumulator kann innerhalb des gleichen Befehls sowohl als Eingabe als auch als Ausgabe (als Quelle und Ziel) angesprochen werden.

Die ALU führt auch *Schieben* und *Rotieren* aus.

Ein Schiebebefehl verschiebt den Inhalt eines Bytes um eine oder mehrere Stellen nach links oder nach rechts. Dies ist in Abb. 2.3 abgebildet. Jedes Bit wurde um eine Position nach links verschoben. Die Einzelheiten von Schieben und Rotieren werden im nächsten Kapitel dargestellt.



Achtung: Einige Schiebe- und Rotier-Operationen schließen das Carry nicht ein.

Abb. 2.3: Schieben und Rotieren

Der Schieber kann am Ausgang der ALU liegen, wie in Abb. 2.2 gezeichnet, er kann aber auch am Eingang liegen.

Links von der ALU liegen die *Flags* oder das *Statusregister*. Ihre Aufgabe ist es, spezielle Zustände innerhalb des Prozessors zu speichern. Der Inhalt der Flags kann mit speziellen Befehlen getestet werden, oder er kann über den internen Datenbus gelesen werden. Ein *bedingter* Befehl kann abhängig vom Wert eines dieser Bits die Ausführung eines neuen Programms bewirken.

Die Rolle des Statusbits im Z80 wird später in diesem Kapitel erklärt.

Flags setzen

Die meisten Befehle, die der Mikroprozessor ausführt, beeinflussen ein oder mehrere Flags. Es ist wichtig, immer auf das Schaubild zurückzugreifen, das der Hersteller mitliefert, und das angibt, welche Bits durch die Befehle beeinflußt werden. Dies ist wesentlich, wenn man verstehen will, wie ein Programm abläuft. Für den Z 80 enthält der Anhang ein solches Schaubild.

Die Register

Dazu wollen wir Abb. 2.2 betrachten. Auf der linken Seite des Bildes erscheinen die Register des Mikroprozessors. Man kann die *Universalregister* und die *Adreßregister* unterscheiden.

Die Universalregister

Universalregister werden für die Aufgabe der ALU benötigt, Daten mit hoher Geschwindigkeit zu verarbeiten. Weil die Zahl der Bits, die inner-

halb eines Befehls sinnvoll ist, beschränkt ist, ist die Zahl der Register (die man direkt ansprechen kann) normalerweise auf weniger als acht beschränkt. Jedes dieser Register ist ein Satz von acht Flip-Flops, die mit dem internen bidirektionalen Datenbus verbunden sind. Diese acht Bits können gleichzeitig vom oder zum Datenbus übertragen werden. Die Ausführung dieser Register als MOS-Flip-Flops bedeutet die schnellste Art von verfügbarem Speicher, auf ihren Inhalt kann man in einigen zehn Nanosekunden zugreifen.

Interne Register werden normalerweise von 0 bis n durchnumeriert. Die Aufgabe der Register ist glücklicherweise nicht festgelegt: man nennt sie „Universalregister". Sie können beliebige Daten, die das Programm verwendet, enthalten.

Diese Universalregister verwendet man normalerweise, um acht Bit breite Daten zu speichern. Bei einigen Mikroprozessoren besteht die Möglichkeit, *zwei* dieser Register gleichzeitig zu ändern. Diese nennt man dann Registerpaare. Diese Einrichtung ermöglicht es, 16 Bit breite Größen (Daten oder Adressen) zu speichern.

Die Adreßregister

Die Adreßregister sind 16-Bit-Register, die zum Speichern von Adressen vorgesehen sind. Man nennt sie auch oft *Zähler* oder *Zeiger*. Ihre wesentliche Eigenschaft ist es, daß sie mit dem Adreßbus verbunden sind. Die Adreßregister beliefern den Adreßbus. Der Adreßbus erscheint im linken und im unteren Teil von Abb. 2.4.

Der einzige Weg, über den der Inhalt dieser Register geladen werden kann, ist der Datenbus. Zwei Transfers über den Datenbus sind nötig, um 16 Bit zu übertragen. Um zwischen der unteren und der oberen Hälf-

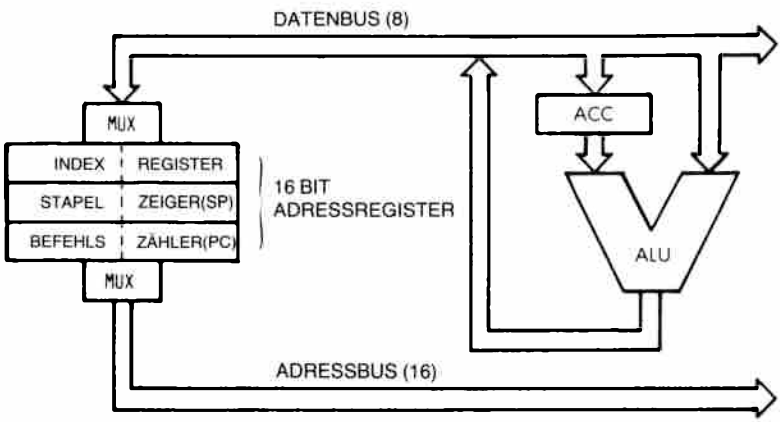


Abb. 2.4: Die 16-Bit-Adreßregister erzeugen den Adreßbus

te jedes Registers zu unterscheiden, werden diese normalerweise mit L (low, untere Hälfte) oder H (high, obere Hälfte) bezeichnet und benennen die Bits 0 bis 7 bzw. die Bits 8 bis 15. Diese Bezeichnung wird immer dann verwendet, wenn man zwischen den beiden Hälften der Register unterscheiden muß. In den meisten Mikroprozessoren gibt es wenigstens zwei Adreßregister. „MUX" in Abb. 2.4 bedeutet Multiplexer.

Befehlszähler (PC)

Der *Befehlszähler* (program counter) muß in jedem Mikroprozessor vorhanden sein. Er enthält die Adresse des Befehls, der als nächster ausgeführt werden soll. Der Befehlszähler ist für die Ausführung eines Programms unerläßlich und grundsätzlich notwendig. Der Mechanismus der Ausführung eines Programms und der automatischen Abfolge der Befehle wird im nächsten Abschnitt erklärt. Kurz gesagt wird ein Programm normalerweise schrittweise abgearbeitet. Um auf den nächsten Befehl richtig zugreifen zu können, ist es notwendig, ihn vom Speicher in den Mikroprozessor zu bringen. Der Inhalt des Befehlszählers wird auf den Adreßbus gelegt und zum Speicher übertragen. Im Speicher wird der Inhalt der Zelle, die durch diese Adresse angesprochen wird, ausgelesen und das entsprechende Wort zur MPU zurückgeschickt. Dies ist der Befehl. Bei einigen wenigen außergewöhnlichen Mikroprozessoren, wie bei dem aus zwei Bausteinen bestehenden F8, gibt es keinen PC im Mikroprozessor. Dies bedeutet nicht, daß das System keinen Befehlszähler enthält. Aus Gründen der Effektivität wird der PC direkt im Speicherbaustein eingebaut.

Stapelzeiger (SP)

Der Stapel (stack) wurde noch nicht eingeführt und wird im nächsten Abschnitt beschrieben. In den meisten leistungsfähigen Mikroprozessoren für universelle Anwendungen ist der Stapel softwaremäßig, d. h. im Speicher realisiert. Um die Position des obersten Stapelelements im Speicher zu verfolgen, hat ein 16-Bit-Register die Aufgabe des *Stapelzeigers.* Der Stapelzeiger enthält die Adresse des Stapelelementes im Speicher. Es wird sich zeigen, daß der Stapel für Interrupts und Unterprogramme unerläßlich ist.

Indexregister (IX)

Die indizierte Adressierung ist eine Möglichkeit der Speicheradressierung, über die Mikroprozessoren nicht immer verfügen. Die verschiedenen Techniken der Adressierung werden in Kapitel 5 beschrieben. Die indizierte Adressierung bietet die Möglichkeit, mit einem einzigen Befehl auf Blöcke von Daten zuzugreifen. Ein *Indexregister* enthält normalerweise eine Distanz, die automatisch zu einer Basisadresse addiert wird (oder es enthält eine Basisadresse, die zu einer Distanz addiert wird). Kurz gesagt, verwendet man die indizierte Adressierung, um auf ein beliebiges Wort in einem Block von Daten zuzugreifen.

Der Stapel

Ein *Stapel* wird formal eine LIFO-Struktur genannt (last-in, first-out). Ein Stapel ist ein Satz von Registern oder Speicherstellen, die dieser Datenstruktur zugeordnet sind. Die wesentliche Eigenschaft des Stapels ist seine *chronologische* Struktur. Das erste Element, das im Stapel abgelegt wird, steht immer am unteren Ende des Stapels. Das Element, das als letztes abgelegt wurde, liegt am oberen Ende des Stapels. Analog funktioniert ein Stapel Teller auf der Theke eines Restaurants. In der Theke ist ein Loch mit einer Feder am Boden. Teller werden in dem Loch aufgestapelt. Mit dieser Anordnung wird gewährleistet, daß der Teller, der als erster abgelegt wurde (der Älteste), immer ganz unten ist. Der Teller, der als nächster abgelegt wurde, liegt darüber. Dieses Beispiel veranschaulicht noch eine andere Eigenschaft des Stapels. Bei normalem Gebrauch kann man mit nur zwei Befehlen auf den Stapel zugreifen: „Push" und „Pop" (oder „Pull"). Mit dem Befehl *Push* wird ein Element oben auf den Stapel gelegt (beim Z80 sind es zwei Elemente). Der Befehl *Pull* entfernt das oberste Element vom Stapel. Bei einem Mikroprozessor wird der *Akkumulator* auf den Stapel abgelegt. Der Befehl *Pop* überträgt das oberste Element des Stapels in den Akkumulator. Zur Übertragung anderer spezieller Register, wie des Statusregisters zum Stapel, kann es spezielle Befehle geben. Der Z80 ist in dieser Hinsicht vielseitiger als die meisten anderen Mikroprozessoren.

Der Stapel wird benötigt, wenn man folgende drei Hilfsmittel bei der Programmierung verwenden will: Unterprogramme, Interrupts und kurzzeitige Datenspeicher.Die Aufgabe des Stapels bei der Abarbeitung von Unterprogrammen wird in Kapitel 3 (Grundlegende Techniken der Programmierung) erklärt. Die Funktion des Stapels bei Interrupts wird in Kapitel 6 (Ein-/Ausgabetechniken) beschrieben. Die Rolle des Stapels bei der schnellen Zwischenspeicherung von Daten wird schließlich an speziellen Programmbeispielen gezeigt werden.

An diesem Punkt nehmen wir einfach an, daß ein Stapel in jedem Computersystem nötig ist. Ein Stapel kann auf zwei Arten realisiert werden:

1. Im Mikroprozessor selbst steht eine feste Zahl von Registern zur Verfügung. Dies ist ein „Hardwarestapel". Sein Vorteil ist die hohe Geschwindigkeit. Er hat jedoch den Nachteil einer begrenzten Anzahl von Stapelelementen.

2. Die meisten universellen Mikroprozessoren verwenden eine andere Methode, den Softwarestapel, um den Stapel nicht auf eine sehr kleine Anzahl von Registern zu beschränken. Dieses Verfahren wurde auch beim Z80 gewählt. Beim Softwareverfahren enthält ein spezielles Register im Mikroprozessor, hier das Register SP, den Stapelzeiger (oder manchmal die Adresse des obersten Elements plus eins). Der Stapel wird dann als Speicherbereich realisiert. Der Stapelzeiger benötigt deshalb 16 Bit, um auf eine beliebige Stelle des Speichers zeigen zu können.

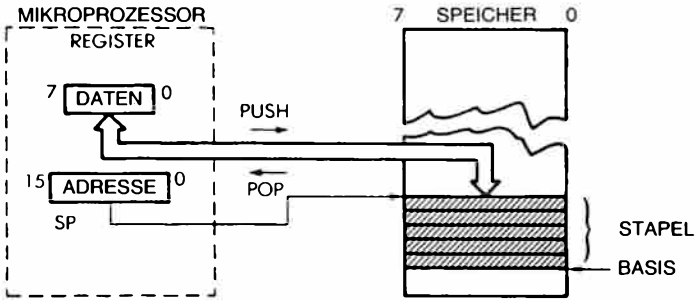


Abb. 2.5: Die beiden Befehle zur Manipulation des Stapels

Der Zyklus der Befehlsausführung

Wir wollen uns jetzt der Abb. 2.6 zuwenden. Die MPU erscheint auf der linken, der Speicher auf der rechten Seite. Der Speicherbaustein kann ein ROM oder ein RAM sein oder ein anderer Baustein, der Speicher enthält. Der Speicher wird verwendet, um Befehle und Daten zu speichern. Hier wollen wir einen Befehl aus dem Speicher holen, um die Rolle des Befehlszählers zu veranschaulichen. Wir wollen annehmen, daß der Inhalt des Befehlszählers gültig ist. Er enthält dann eine 16-Bit-Adresse, die Adresse des nächsten Befehls, die aus dem Speicher geholt werden soll. Jeder Prozessor arbeitet in drei Zyklen:

1 – Hole den nächsten Befehl (Fetch)
2 – Dekodiere den Befehl (Decode)
3 – Führe den Befehl aus (Execute).

Hole Befehl

Wir wollen jetzt den Ablauf verfolgen. Im ersten Zyklus wird der Inhalt des Befehlszählers auf den Adreßbus gelegt und zum Speicher übertragen (über den Adreßbus). Wenn nötig, wird gleichzeitig ein Lesesignal (read) auf den Steuerbus des Systems ausgegeben. Der Speicher empfängt die Adresse. Diese Adresse wird verwendet, um eine Speicherzelle im Speicher auszuwählen. Nachdem der Speicher die Adresse empfangen hat, wird die Adresse von internen Dekodern dekodiert und wählt die Speicherzelle an, die durch die Adresse angegeben wurde. Einige hundert Nanosekunden später gibt der Speicher die acht Bit Daten, die durch die angegebene Adresse ausgewählt wurden, auf den Datenbus aus. Dieses 8-Bit-Wort ist der Befehl, den wir hereinholen wollen. In unserer Abbildung wird dieser Befehl oben auf den Datenbus gelegt.

Wir wollen die Abfolge kurz zusammenfassen: Der Inhalt des Befehlszählers wird auf den Adreßbus ausgegeben. Ein Lesesignal wird erzeugt. Um den *Speicherzyklus* und vielleicht 300 Nanosekunden verzögert wird der Befehl an der angegebenen Adresse auf den Datenbus gelegt (vor-

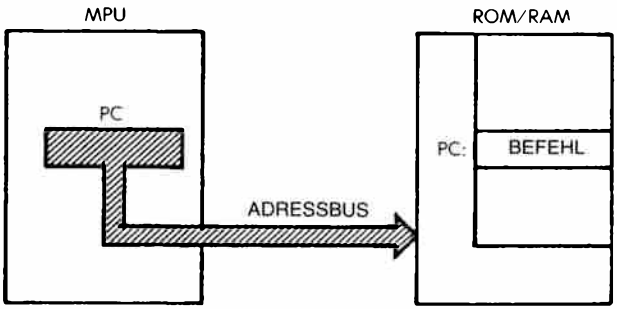


Abb. 2.6: Einen Befehl aus dem Speicher holen

ausgesetzt, es ist ein Ein-Byte-Befehl). Der Mikroprozessor liest dann den Datenbus und legt seinen Inhalt in einem speziellen internen Register, dem IR Register ab. Das IR Register ist das *Befehlsregister* (instruction register). Es ist acht Bit breit und wird dazu verwendet, den gerade aus dem Speicher geholten Befehl aufzunehmen. Der Fetch-Zyklus ist damit abgeschlossen. Die 8 Bit des Befehls liegen jetzt in einem speziellen Register in der MPU, dem IR Register. Das IR erscheint auf der linken Seite der Abb. 2.7. Es ist für den Programmierer nicht zugänglich.

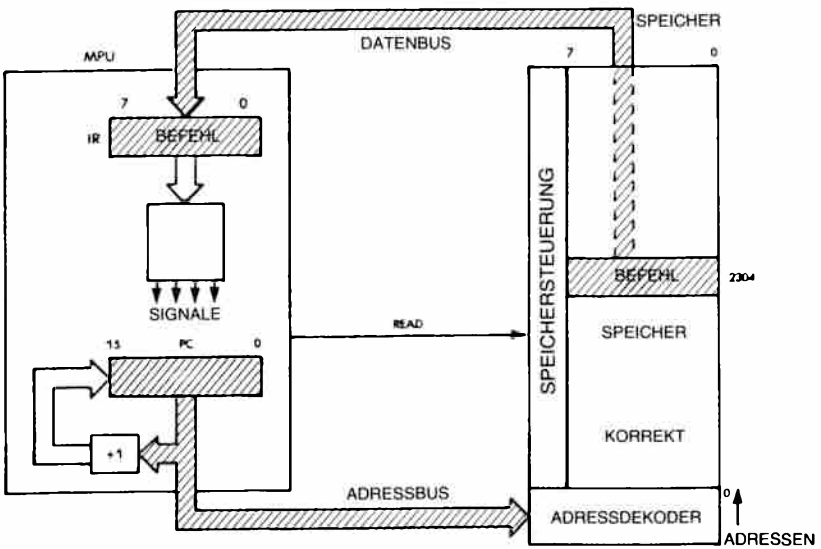


Abb. 2.7: Automatische Abfolge

Dekodierung und Ausführung

Sobald der Befehl im IR steht, dekodiert die Steuereinheit seinen Inhalt und erzeugt die Folge von internen und externen Signalen, die zur Ausführung dieses Befehls nötig sind. Es tritt deshalb eine kurze Verzögerung auf, während der der Befehl dekodiert wird, gefolgt von der Phase der Ausführung, deren Länge von der Art des Befehls abhängt. Einige Befehle laufen vollständig innerhalb der MPU ab, andere Befehle holen Daten vom Speicher oder legen sie dort ab. Deshalb benötigen unterschiedliche Befehle verschieden lange Ausführungszeiten. Diese Zeitdauer wird als Anzahl von (Takt-) Zyklen angegeben. Die Zahl der Zyklen für jeden Befehl ist im Anhang angegeben. Weil verschiedene Taktfrequenzen verwendet werden können, gibt man die Ausführungszeit normalerweise als Anzahl von Taktzyklen an und nicht in Nanosekunden.

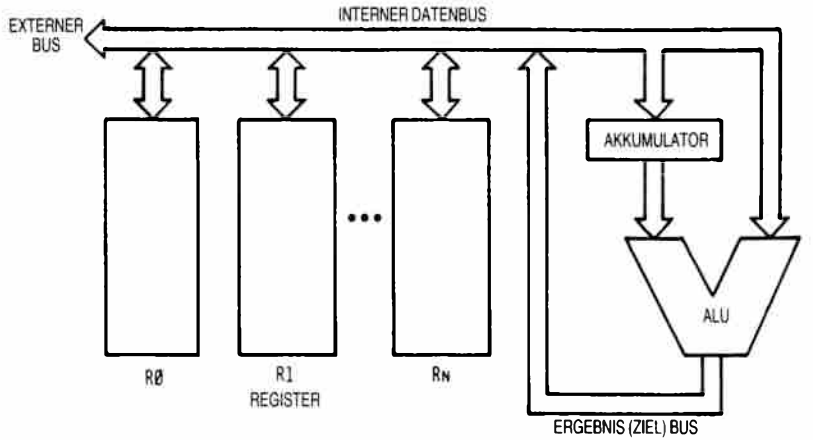


Abb. 2.8: Architektur mit einem einzelnen Bus

Hole nächsten Befehl

Wir haben jetzt unter Verwendung des Befehlszählers beschrieben, wie ein Befehl aus dem Speicher gelesen werden kann. Während der Ausführung eines Programms werden Befehle *nacheinander* aus dem Speicher geholt. Deshalb muß ein automatischer Mechanismus vorgesehen sein, um die Befehle der Reihe nach hereinzuholen. Diese Aufgabe erfüllt ein einfacher Inkrementierer, der an den Befehlszähler angeschlossen ist. Dies wird in Abb. 2.7 veranschaulicht. Jedes Mal, wenn der Inhalt des Befehlszählers (in der Zeichnung unten) auf den Adreßbus ausgegeben wird, wird sein Inhalt inkrementiert und in den Befehlszähler zurückgeschrieben. Wenn der Befehlszähler beispielsweise den Wert „0" enthielt, wurde die „0" auf den Adreßbus ausgegeben. Dann wird der Inhalt des Befehlszählers inkrementiert und der Wert „1" in den Be-

fehlszähler zurückgeschrieben. Mit dieser Methode wird beim nächsten Mal, wenn der Befehlszähler verwendet wird, der Befehl an der Adresse 1 geholt. Wir haben so einen *automatischen Mechanismus zur Befehlsabfolge* eingebaut.

Es muß betont werden, daß die obige Beschreibung vereinfacht ist. In Wirklichkeit können einige Befehle zwei oder drei Byte lang sein, so daß aufeinanderfolgende Bytes auf diese Art vom Speicher geholt werden. Der Mechanismus bleibt jedoch identisch. Der Befehlszähler dient genauso dazu, aufeinanderfolgende Bytes eines Befehls zu holen, wie auf-

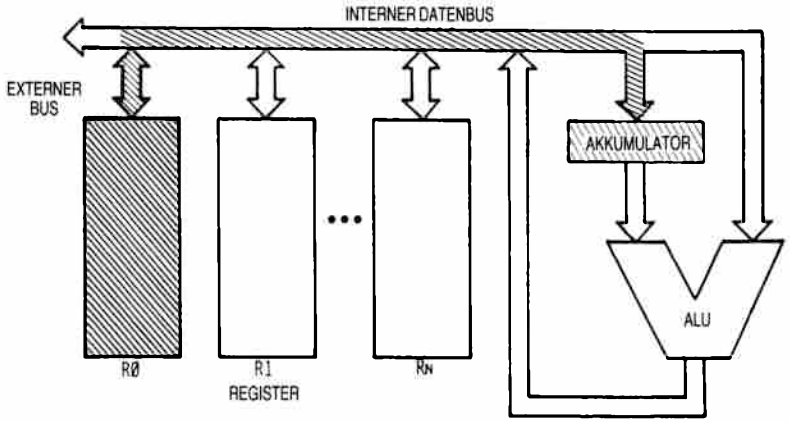


Abb. 2.9: Ausführung einer Addition – R0 in den Akku)

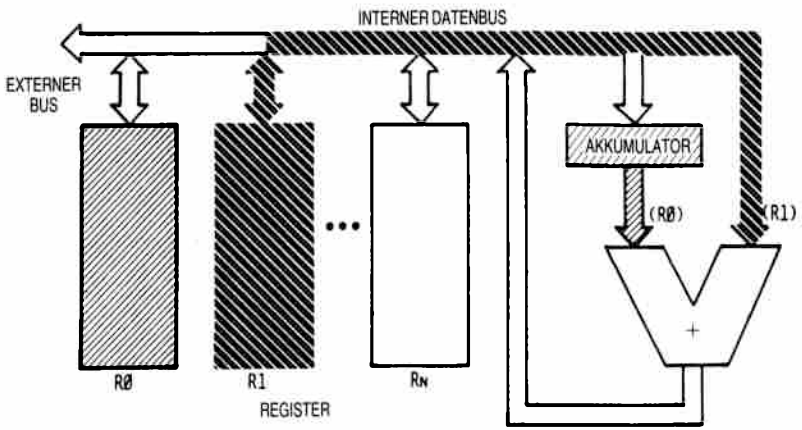


Abb. 2.10: Addition – Zweites Register R1 in die ALU

einanderfolgende Befehle. Der Befehlszähler bildet zusammen mit dem Inkrementierer einen automatischen Mechanismus mit dem Zweck, auf aufeinanderfolgende Speicherstellen zu zeigen.

Wir werden jetzt einen Befehl innerhalb der MPU abarbeiten (siehe Abb. 2.8). Ein typischer Befehl ist beispielsweise: R0 = R0 + R1. Dies heißt: Addiere die Inhalte von R0 und R1 und speichere das Ergebnis in R0. Um diese Operation auszuführen, wird der Inhalt von R0 aus dem Register R0 gelesen, über den Bus zum linken Eingang der ALU übertragen und in dem Pufferregister dort gespeichert. Dann wird R1 angewählt und sein Inhalt wird auf den Bus ausgelesen und zum rechten Eingang der ALU übertragen. Dieser Ablauf ist in den Abb. 2.9 und 2.10 veranschaulicht. Jetzt wird der rechte Eingang der ALU von R1 und der linke Eingang von dem Pufferregister bestimmt, das den vorhergehenden Wert von R0 enthält. Die Operation kann jetzt durchgeführt werden. Die Addition wird von der ALU ausgeführt und das Ergebnis erscheint am Ausgang der ALU, in Abb. 2.11 in der rechten unteren Ecke. Das Ergebnis wird auf den Bus ausgegeben und zum Register R0 zurückübertragen. Das heißt, daß die Eingabesteuerung von R0 freigegeben wird, so daß Daten eingeschrieben werden können. Der Befehl ist jetzt vollständig ausgeführt worden. Das Ergebnis der Addition steht in R0. Man sollte beachten, daß der Inhalt von R1 durch diese Operation nicht verändert wurde. Dies ist ein allgemeines Prinzip: Der Inhalt eines Registers oder eines Schreib-/Lesespeichers wird durch eine Leseoperation nicht verändert.

Das Pufferregister am linken Eingang der ALU war notwendig, um den Inhalt von R0 *zwischenzuspeichern*, so daß der einzige Bus für eine andere Übertragung verwendet werden kann. Ein Problem bleibt jedoch bestehen.

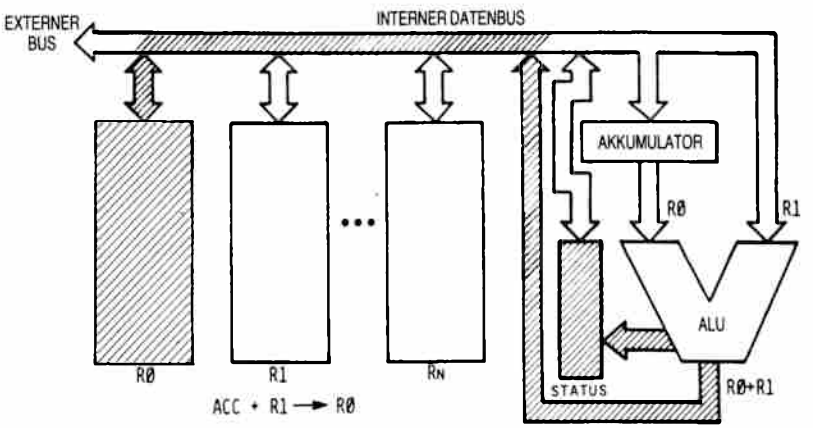


Abb. 2.11: Das Ergebnis wird erzeugt und kommt nach R0.

Der kritische Wettlauf

Der einfache Aufbau, der in Abb. 2.8 gezeigt wurde, funktioniert nicht richtig.

Frage: *Wo liegt das zeitlich kritische Problem:*

Antwort: Das Problem besteht darin, daß das Ergebnis, das die ALU ausgibt, auf den Bus ausgegeben wird. Es breitet sich nicht nur in der Richtung nach R0 aus, sondern entlang des ganzen Busses. Es wird speziell den rechten Eingang des ALU verändern und damit auch das Er-

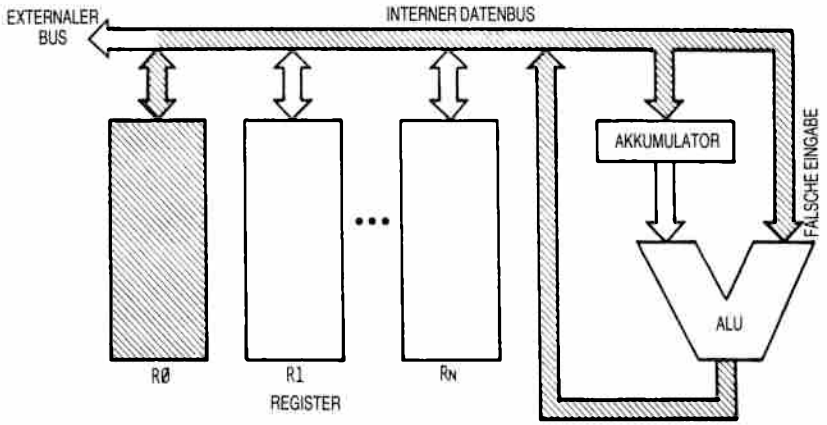


Abb. 2.12: Der kritische Wettlauf

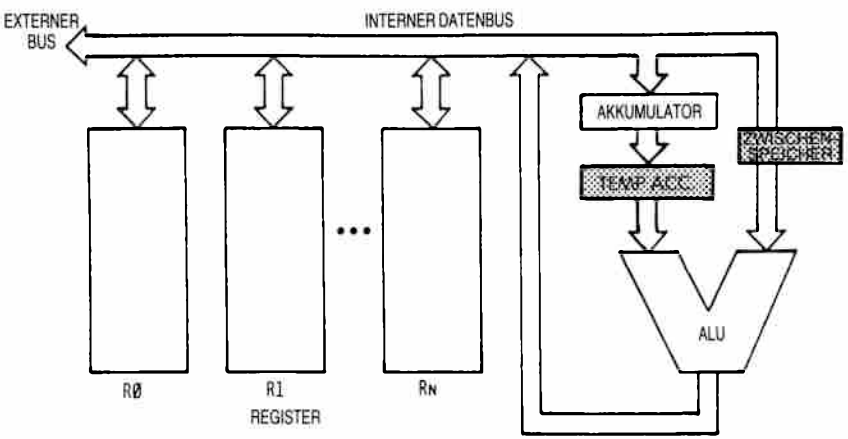


Abb. 2.13: Zwei Puffer werden benötigt

gebnis, das einige Nanosekunden später herauskommt. Das ist der *kritische Wettlauf*. Der Ausgang der ALU muß von ihrem Eingang getrennt werden (siehe Abb. 2.12).

Es gibt verschiedene Möglichkeiten, die den Eingang der ALU von ihrem Ausgang trennen. Man könnte ein Pufferregister verwenden. Das Pufferregister könnte am Ausgang oder am Eingang der ALU angebracht werden. Üblicherweise bringt man es am Eingang der ALU an. Hier würde es am rechten Eingang angeschlossen. Die Pufferung des Systems reicht jetzt für ein korrektes Arbeiten aus. Später in diesem Kapitel wird gezeigt werden, daß das Register am linken Eingang dieser Abbildung als Akkumulator verwendet wird (vorausgesetzt, man verwendet ein Byte lange Befehle), dann benötigt der Akku ebenfalls einen Puffer, wie in Abb. 2.13 dargestellt.

Interne Organisation des Z80

Die Begriffe, die man braucht, um die internen Elemente des Mikroprozessors zu verstehen, wurden bereits eingeführt. Wir wollen jetzt den Z80 selbst genauer untersuchen und seine Fähigkeiten beschreiben. Die interne Organisation des Z80 ist in Abb. 2.14 dargestellt. Das Diagramm zeigt eine logische Beschreibung des Bausteins. Es mag zusätzliche Verbindungen geben, die aber nicht eingezeichnet sind. Wir wollen das Diagramm von rechts nach links untersuchen.

Auf der rechten Seite des Bildes erkennt man die *Arithmetik- und Logikeinheit* (ALU) an ihrer charakteristischen „V"-Form. Der Akkumulator, der im vorhergehenden Abschnitt beschrieben wurde, ist mit A im rechten Eingangspfad der ALU bezeichnet. Im vorhergehenden Abschnitt wurde gezeigt, daß der Akkumulator mit einem *Pufferregister* verbunden werden sollte. Dieses Register ist mit ACT bezeichnet. Auch der linke Eingang der ALU ist mit einem *Zwischenspeicher* versehen, der TMP genannt wird. Die Arbeitsweise der ALU wird im nächsten Abschnitt klar werden, in dem wir die Ausführung tatsächlicher Befehle beschreiben werden.

Das *Flag-Register* wird beim Z80 „F" genannt und ist rechts vom Akkumulator eingezeichnet. Der Inhalt des FLAG-Registers wird im wesentlichen von der ALU bestimmt, es wird aber gezeigt werden, daß einzelne Bits auch durch andere Baugruppen oder Ereignisse verändert werden können.

Der Akkumulator und die Flag-Register sind als Doppelregister gezeichnet und entsprechend A, A' und F, F' genannt. Der Grund dafür ist, daß der Z80 intern mit zwei Registersätzen ausgestattet ist: A + F und A' + F'. Man kann jedoch immer nur *einen* dieser Registersätze verwenden. Mit einem speziellen Befehl können die Inhalte von A und F mit denen von A' und F' vertauscht werden. Um die Erklärungen zu vereinfachen, werden in den meisten der folgenden Diagramme nur A und

F dargestellt. Der Leser sollte aber daran denken, daß er wahlweise auf den Zweitregistersatz A' und F' umschlagen kann.

Die Rolle der einzelnen Flags im Flag-Register wird in Kapitel 3 (Grundlegende Techniken der Programmierung) beschrieben.

In der Mitte der Abbildung ist ein großer Block von Registern eingezeichnet. Im oberen Teil des Registerblocks kann man zwei identische Gruppen erkennen. Jede enthält sechs Register mit den Bezeichnungen B, C, D, E, H, L. Dies sind die *„Acht-Bit-Universalregister* des Z80. Hinzu kommt ein internes Registerpaar (W, Z), das dem Programmierer jedoch nicht zugänglich ist. Es wird hier nur erwähnt, da dieses Registerpaar für die Ausführung von Befehlen innerhalb des Z80 von Bedeutung ist (vergl. S. 65 ff.).

Zwei Besonderheiten unterscheiden den Z80 von dem Standardmikroprozessor, der am Anfang dieses Kapitels beschrieben wurde. Zum einen besitzt der Z80 *zwei* Blöcke von Registern, d.h. zwei identische Gruppen von sechs Registern. Es können allerdings zu einem Zeitpunkt immer nur sechs Register verwendet werden. Es gibt jedoch spezielle Befehle, um zwischen den Registerblöcken hin und her zu schalten. Ein Block verhält sich deshalb wie ein interner Speicher, während der andere ein arbeitender Satz von Registern ist. Mögliche Anwendungen dieser speziellen Eigenschaften werden im nächsten Kapitel beschrieben.

Für den Augenblick werden wir annehmen, daß es nur die sechs Universalregister B, C, D, E, H. L gibt, den Zweitregistersatz beachten wir vorläufig nicht, um Verwirrung zu vermeiden.

Das Symbol MUX oberhalb des Registerblocks ist eine Abkürzung für Multiplexer. Die Daten, die vom internen Datenbus hereinkommen, werden durch den Multiplexer zu dem ausgewählten Register geleitet. Zu einem Zeitpunkt kann jedoch nur eines dieser Register mit dem internen Datenbus verbunden werden.

Eine zweite Eigenschaft dieser sechs Register ist es, zusätzlich zu ihrer Eigenschaft, Universalregister zu sein, daß sie eine Verbindung zum *Adressbus* besitzen. Deshalb sind sie zu *Paaren* zusammengefaßt. Es können beispielsweise die Inhalte der Register B und C gleichzeitig auf den 16-Bit-Adreßbus ausgegeben werden, der im unteren Teil der Abbildung dargestellt ist. So kann dieser Block von sechs Registern entweder verwendet werden, um 8-Bit-Daten zu speichern, oder als *16-Bit-Zeiger* zur Speicheradressierung.

Die dritte Gruppe von Registern, die unterhalb der vorher beschriebenen Register in der Mitte von Abb. 2.14 eingezeichnet ist, enthält vier „reine" Adreßregister. Wie in jedem Mikroprozessor finden wir den Befehlszähler (PC) und den Stapelzeiger (SP). Erinnern Sie sich daran , daß der Befehlszähler die Adresse des Befehls enthält, der als nächster ausgeführt werden soll.

Der Stapelzeiger zeigt auf das oberste Element des Stapels im Speicher. Beim Z80 zeigt er auf den letzten Eintrag in den Stapel. (Bei anderen Mikroprozessoren zeigt er gerade über den letzten Eintrag.) Außerdem wächst der Stapel *nach unten*, das heißt zu kleineren Adressen hin.

Das bedeutet, daß der Stapelzeiger jedesmal *dekrementiert* werden muß, wenn ein neues Wort auf den Stapel *abgelegt* (Push) wurde. Umgekehrt muß der Stapelzeiger *inkrementiert* werden, wenn ein Wort vom Stapel *geholt* (Pop) wurde. Beim Z80 erfassen Push und Pop jeweils *zwei* Worte gleichzeitig, so daß der Stapelzeiger um zwei dekrementiert bzw. inkrementiert wird.

Wenn wir die beiden verbleibenden Register aus dieser Vierergruppe betrachten, finden wir einen neuen Typ von Register, der noch nicht beschrieben wurde: zwei *Indexregister,* genannt IX (Indexregister X) und IY (Indexregister Y). Diese beiden Register sind mit einem speziellen Addierer verbunden, der als kleine V-förmige ALU rechts von diesen Registern in Abb. 2.14 eingezeichnet ist. Ein Byte, das über den internen Datenbus hereinkommt, kann zum Inhalt von IX oder IY addiert werden. Dieses Byte nennt man *Distanz* (displacement), wenn man einen indizierten Befehl verwendet. Es gibt spezielle Befehle, die diese Distanz automatisch zum Inhalt von IX oder IY addieren und eine Adresse erzeugen. Diesen Vorgang nennt man *Indizieren.* Er erlaubt einen vorteilhaften Zugriff auf irgendeinen Block fortlaufender Daten. Diese nützliche Fähigkeit wird in Kapitel 5 bei den Adressierungstechniken beschrieben.

Schließlich erscheint links unterhalb des Registerblocks ein spezielles Kästchen, das mit „+ −1" markiert ist. Dies ist ein Inkrementierer/Dekrementierer. Der Inhalt jedes der vier Register, die zu dem zuletzt beschriebenen Block gehören (die reinen Adreßregister), kann immer dann automatisch inkrementiert oder dekrementiert werden, wenn das Register eine Adresse auf den internen Datenbus ausgibt. Dies ist eine wichtige Möglichkeit, wenn man automatische *Programmschleifen* einbauen will, was im nächsten Kapitel beschrieben wird. Wenn man diese Möglichkeit ausnutzt, kann man bequem auf aufeinanderfolgende Speicherzellen zugreifen.

Jetzt wollen wir uns dem linken Teil der Abbildung zuwenden. Ein einzelnes Registerpaar ist auf der linken Seite eingezeichnet: I und R. Das I-Register nennt man *Interruptregister.* Seine Funktion wird im Abschnitt über Interrupts in Kapitel 6 (Ein-/Ausgabetechniken) beschrieben. Es wird nur in einer speziellen Betriebsart verwendet, wo ein indirekter Unterprogrammaufruf als Antwort auf einen Interrupt erzeugt wird. Das I-Register speichert die obere Hälfte der indirekten Adresse. Die untere Hälfte wird von dem Baustein, der den Interrupt auslöst, geliefert.

Das R-Register ist das *Refreshregister.* Es dient dazu, dynamische Speicher automatisch aufzufrischen. Ein solches Register lag bisher üblicher-

weise außerhalb des Mikroprozessors, weil es zu dem dynamischen Speicher gehört. Es ist eine vorteilhafte Einrichtung, die den Aufwand an zusätzlicher Hardware für einige Typen dynamischer Speicher verkleinert. Es wird hier nicht zum Zweck der Programmierung verwendet werden, da es im wesentlichen eine Hardwareeinrichtung ist (siehe das Buch „Mikroprozessor Interface Techniken" für eine detaillierte Beschreibung des Auffrischens von Speichern). Es kann jedoch beispielsweise als Softwareuhr verwendet werden.

Wir wollen uns jetzt dem Teil der Zeichnung ganz links zuwenden. Hier ist das Steuerwerk des Mikroprozessors untergebracht. Von oben nach unten finden wir zuerst das *Befehlsregister* IR, das den Befehl enthält, der ausgeführt werden soll. Das Register IR ist völlig verschieden von dem Registerpaar „I, R", das wir oben beschrieben haben. Der Befehl wird vom Speicher auf den Datenbus ausgegeben, über den internen Datenbus übertragen und schließlich im Befehlsregister abgelegt. Unter dem Befehlsregister erscheint der *Dekoder*, der Signale zum Controller/Sequencer schickt und die Ausführung des Befehls innerhalb und außerhalb des Mikroprozessors veranlaßt. Das *Steuerwerk* erzeugt und verwaltet den Kontrollbus, der im unteren Teil der Zeichnung erscheint.

Die drei Busse, die vom System verwaltet oder erzeugt werden, d. h. Datenbus, Adreßbus und Kontrollbus, setzen sich über die Anschlüsse des Mikroprozessors nach außen fort. Die Verbindungen nach außen sind in der Abbildung ganz rechts eingezeichnet. Die inneren Busse sind von den äußeren durch Puffer getrennt, wie in Abb. 2.14 dargestellt.

Jetzt sind alle logischen Elemente des Z80 erklärt worden. Es ist nicht notwendig, die Arbeitsweise des Z80 im einzelnen zu verstehen, wenn man anfangen will, Programme zu schreiben. Für den Programmierer jedoch, der effizienten Kode schreiben will, hängen Geschwindigkeit und Länge eines Programms von der richtigen Wahl der Register ebenso ab, wie von der richtigen Anwendung der Techniken. Um hier richtig zu entscheiden, muß man verstehen, wie Befehle innerhalb des Mikroprozessors abgearbeitet werden. Wir werden deshalb hier die Ausführung typischer Befehle im Innern des Z80 untersuchen und die Aufgaben und die Verwendung der internen Register und Busse zeigen.

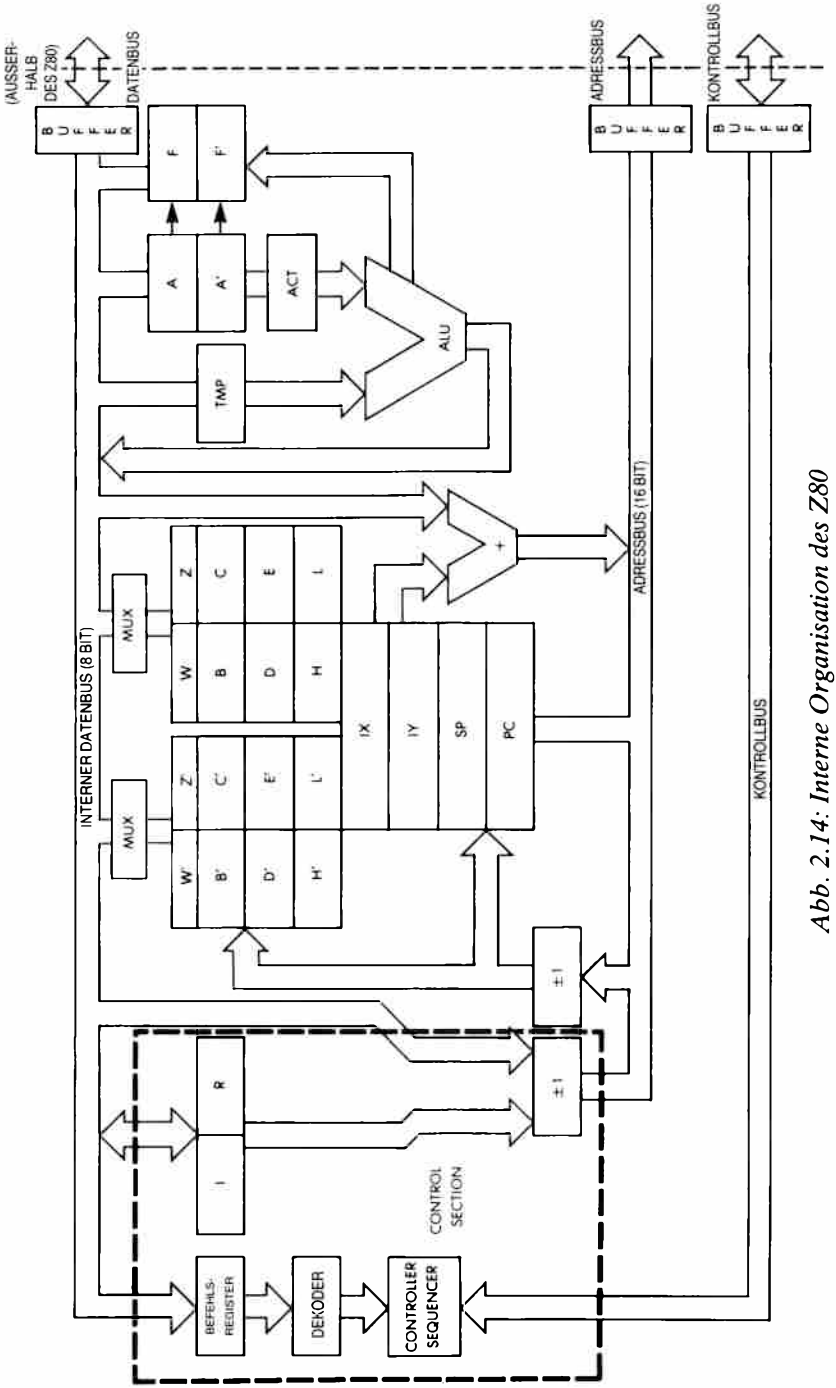


Abb. 2.14: Interne Organisation des Z80

Befehlsformate

Die Befehle des Z80 sind in Kapitel 4 aufgelistet. Z80-Befehle können ein, zwei, drei oder vier Bytes lang sein. Ein Befehl legt fest, welche Operation der Mikroprozessor ausführen soll. Vereinfacht gesagt, kann man jeden Befehl darstellen als einen Operationskode (Opcode), auf den wahlweise ein Operanden- oder Adreßfeld folgen kann. Der Opcode legt fest, welche Operation ausgeführt werden soll. Wenn man die Computersprache streng auslegt, dann gehören zum Operationskode nur die Bits, die festlegen, welche Operation durchgeführt wird, aber nicht die Zeiger auf Register, die eventuell dazu notwendig sind. Bei den Mikroprozessoren versteht man unter dem Opcode aber nicht nur diesen Teil, sondern auch eventuelle Zeiger auf Register, die darin enthalten sein können. Dieser „allgemeinere Opcode" soll aus Gründen der Leistungsfähigkeit in ein Acht-Bit-Wort passen (dies ist der begrenzende Faktor für die Zahl der Befehle, die in einem Mikroprozessor verfügbar sind).

Der 8080 verwendet Befehle, die ein, zwei oder drei Bytes lang sein können (siehe Abb. 2.15). Der Z80 ist jedoch mit zusätzlichen indizierten Befehlen ausgestattet, die ein zusätzliches Byte belegen. Beim Z80 sind die Opcodes im allgemeinen ein Byte lang, mit Ausnahme spezieller Befehle, die einen Zwei-Byte-Opcode belegen.

Einige Befehle verlangen, daß dem Opcode ein Datenbyte folgt. In diesem Fall ist der Befehl ein Zwei-Byte-Befehl, dessen zweites Byte aus Daten besteht (außer bei indizierten Befehlen, die ein weiteres Byte belegen).

In anderen Fällen mag der Befehl eine Adresse benötigen. Eine Adresse nimmt 16 Bit ein und damit zwei Bytes. In diesem Fall ist der Befehl eine Drei-Byte- oder ein Vier-Byte-Befehl.

Für jedes Byte des Befehls muß die Steuereinheit eine Leseoperation ausführen, die vier Taktzyklen in Anspruch nimmt. Je kürzer der Befehl ist, um so schneller wird er ausgeführt.

Ein Ein-Byte-Befehl

Ein-Wort-Befehle sind im Prinzip am schnellsten und werden deshalb von den Programmierern bevorzugt. Ein typischer solcher Befehl beim Z80 ist:

```
LD r,r'
```

Dieser Befehl heißt: Übertrage den Inhalt des Registers r' nach r. Dies ist eine typische „Register-Register"-Operation. Jeder Mikroprozessor muß über solche Befehle verfügen, die es dem Programmierer erlauben, Information aus einem beliebigen Register der Maschine in ein anderes zu übertragen. Befehle, die sich auf spezielle Register beziehen, wie auf den Akkumulator oder auf andere Spezialregister, können einen speziellen Opcode haben.

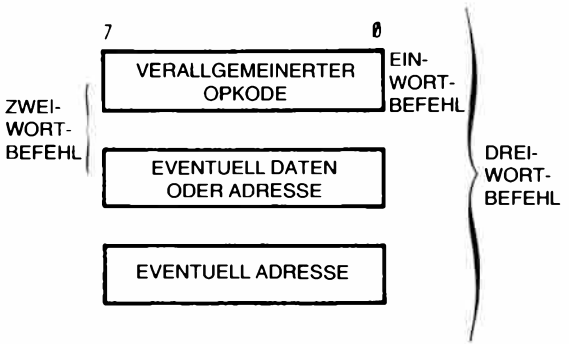


Abb. 2.15: Typisches Befehlsformat

Nach Ausführung des obigen Befehls ist der Inhalt von r gleich dem Inhalt von r'. Der Inhalt von r' wurde durch die Leseoperation nicht verändert.

Jeder Befehl muß intern in binärem Format dargestellt werden. Die obige Darstellung „LD r,r'" ist symbolisch oder *mnemotechnisch*. Sie wird als Darstellung des Befehls in *Assemblersprache* bezeichnet und bedeutet einfach eine vorteilhafte symbolische Form für den Binärkode des Befehls. Der Binärkode für diesen Befehl innerhalb des Speichers ist: 0 1 D D D S S S (Bit 0 bis 7).

Diese Darstellung ist immer noch teilweise symbolisch zu verstehen. Jeder der Buchstaben S und D steht für ein binäres Bit. Die drei Ds „D D D" stehen für die drei Bits, die das *Zielregister* bezeichnen. Mit drei Bit kann man eines von acht möglichen Registern auswählen. Die Kodes für diese Register stehen in Abb. 2.16. Der Kode für das Register B ist beispielsweise „0 0 0", der Kode für das Register C ist „0 0 1" usw. Ebenso stellt „S S S" die drei Bit dar, die auf das *Quellregister* zeigen. Vereinbarungsgemäß ist r' die Quelle und r das Ziel. Die einzelnen Bits in der Binärdarstellung sind nicht so plaziert, wie es für den Programmierer am bequemsten ist, sondern so, wie es für die Steuereinheit des Mikroprozessors, die den Befehl dekodieren und ausführen muß, am geeignetsten ist. Dagegen ist die Darstellung in *Assemblersprache* für den Programmierer gemacht. Man könnte vermuten, daß LD r,r' heißen sollte: Übertrage den Inhalt von r nach r'. Jedoch wurde die Konvention so festgelegt, daß sie in diesem Fall mit der Binärdarstellung übereinstimmt. Diese Festlegung ist natürlich rein willkürlich.

Aufgabe 2.1: Schreibe den Binärkode auf, der den Inhalt von Register B transferiert. Entnimm die Kodes für die Register B und C der Abb. 2.16.

Ein anderes einfaches Beispiel für einen Ein-Wort-Befehl ist:

ADD A,r

Dieser Befehl bewirkt die Addition eines bestimmten Registers (r) in

den Akkumulator (A). Diese Operation kann man symbolisch als A = A + r schreiben. In Kapitel 4 kann man überprüfen, daß die Binärdarstellung dieses Befehls

1 0 0 0 0 S S S

ist, wobei S S S das Register festlegt, das zum Akkumulator addiert werden soll. Für den Registerkode gilt ebenfalls Abb. 2.16.

Aufgabe 2.2: Wie sieht der Binärcode des Befehls aus, der den Inhalt des Registers zum Akkumulator addiert?

KODE	REGISTER
0 0 0	B
0 0 1	C
0 1 0	D
0 1 1	E
1 0 0	H
1 0 1	L
1 1 0	- (SPEICHER)
1 1 1	A

Abb. 2.16: Die Kodes der Register

Ein Zwei-Wort-Befehl

ADD A,n

Dieser einfache Zwei-Wort-Befehl addiert den Inhalt des zweiten Bytes zum Akkumulator. Den Inhalt des zweiten Befehlswortes nennt man ein „Literal". Dies sind Daten, die als acht Bit ohne spezielle Bedeutung behandelt werden. Sie könnten ein Zeichen oder numerische Daten darstellen. Für die Operation ist dies bedeutungslos. Der Befehlskode lautet:

1 1 0 0 0 1 1 0, gefolgt von dem Byte „n".

Man nennt dies eine *unmittelbare* Operation. „Unmittelbar" bedeutet in den meisten Programmiersprachen, daß das nächste Wort oder die nächsten Worte innerhalb des Befehls Daten sind, die nicht (wie ein Opcode) *interpretiert* werden sollen. Das heißt, das nächste oder die beiden nächsten Worte werden als Literals betrachtet.

Das Steuerwerk erkennt, aus wievielen Worten jeder Befehl besteht. Es wird deshalb bei jedem Befehl die richtige Anzahl von Worten holen und ausführen. Je länger jedoch die maximale Wortlänge eines Befehls ist, um so komplexer ist die Dekodierung für das Steuerwerk.

Ein Drei-Wort-Befehl

LD a,(nn)

Dieser Befehl belegt drei Worte. Er bedeutet: Lade den Akkumulator mit dem Inhalt der Speicherzelle, deren Adresse in den nächsten beiden Bytes des Befehls steht. Da eine Adresse 16 Bit lang ist, nimmt sie zwei Worte ein. Dieser Befehl wird binär dargestellt durch:

0 0 1 1 1 0 1 0:	Acht Bit für den Operationskode
Untere Adresse:	Acht Bit für den unteren Teil der Adresse
Obere Adresse:	Acht Bit für den oberen Teil der Adresse

Ausführung von Befehlen innerhalb des Z80

Wir haben gesehen, daß alle Befehle in drei Phasen ausgeführt werden: Holen, Dekodieren, Ausführen. Wir müssen jetzt einige Definitionen einführen. Jede dieser Phasen benötigt mehrere Taktzyklen. Der Z80 führt jede Phase in einem oder mehreren logischen Zyklen aus, die „Maschinenzyklus" genannt werden. Der kürzeste Maschinenzyklus belegt drei Taktzyklen.

Ein Speicherzugriff beansprucht vier Taktzyklen. Da jeder Befehl zunächst aus dem Speicher geholt werden muß, ist der schnellste Befehl vier Taktzyklen lang. Die meisten Befehle dauern länger.

Jeder Maschinenzyklus wird mit M1, M2 uw. bezeichnet und belegt drei oder mehr Taktzyklen oder „Zustände", die man T1, T2 usw. nennt.

Die Holphase

Die Holphase eines Befehls wird während der ersten drei Zustände des Maschinenzyklus M1 ausgeführt: die Taktzyklen heißen T1, T2 und T3. Diese drei Zustände sind bei allen Befehlen des Mikroprozessors gleich, da alle Befehle vor der Ausführung gelesen werden müssen. Der Holmechanismus funktioniert folgendermaßen:

T1: Ausgabe von PC

Die Aufgabe des ersten Schrittes ist es, die Adresse des nächsten Befehls an den Speicher auszugeben. Diese Adresse steht im Befehlszähler (PC). Als erster Schritt beim Hereinholen jedes Befehls wird der Inhalt von PC auf den Adreßbus ausgegeben (siehe Abb. 2.17). Jetzt wird dem Speicher eine Adresse übergeben, und die Adreßdekoder des Speichers dekodieren diese Adresse, um die entsprechende Speicherstelle auszuwählen. Es vergehen mehrere hundert Nanosekunden (eine Nanosekunde ist 10^{-9} Sekunden) bis der Inhalt der ausgewählten Speicherstelle an den Speicherausgängen, die mit dem Datenbus verbunden sind, zur Verfügung steht. Üblicherweise wird ein Computer so entworfen, daß die Speicherlesezeit für eine Operation innerhalb des Mikroprozessors genutzt wird. Diese Operation ist das Inkrementieren des Befehlszählers:

T2: PC = PC + 1

Während der Speicherinhalt ausgelesen wird, wird der Inhalt von PC um 1 inkrementiert (siehe Abb. 2.18). Am Ende von Zustand T2 steht der Speicherinhalt zur Verfügung und kann in den Mikroprozessor übertragen werden:

T3: Befehl ins IR

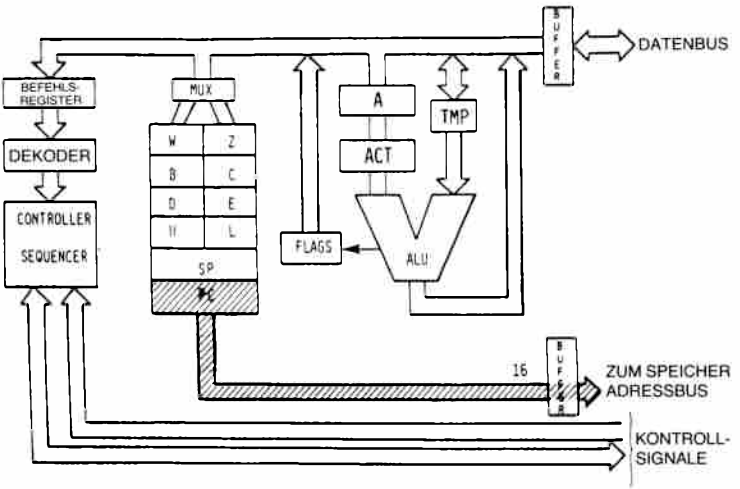


Abb. 2.17: Instruction Fetch – (PC) wird zum Speicher geschickt.

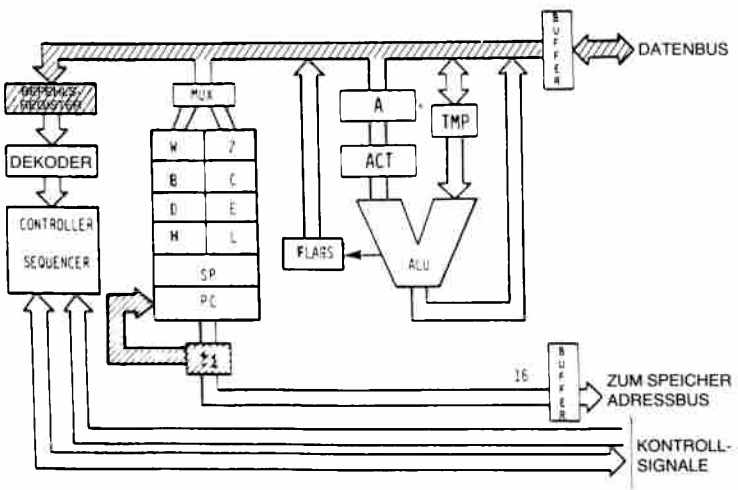


Abb. 2.18: PC wird inkrementiert

Die Dekodier- und die Ausführungsphase

Während des Zustand T3 liegt der Befehl, der aus dem Speicher ausgelesen wurde, auf dem Datenbus und wird in das Befehlsregister des Z80 übertragen, wo er dekodiert wird.

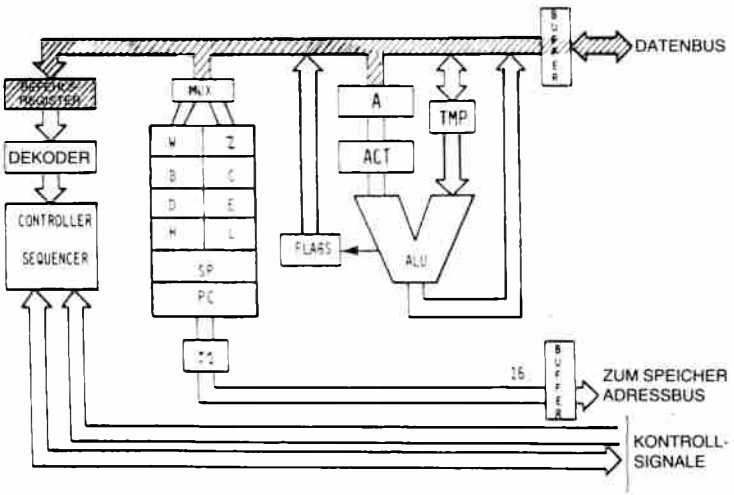


Abb. 2.19: Der Befehl kommt aus dem Speicher ins IR

Es sollte darauf hingewiesen werden, daß der Zustand T4 von M1 immer benötigt wird. Sobald der Befehl während T3 ins IR abgelegt wurde, ist es nötig, ihn zu *dekodieren* und *auszuführen*. Dies beansprucht wenigstens einen weiteren Taktzyklus, T4.

Einige wenige Befehle brauchen einen zusätzlichen Zustand während M1 (Zyklus T5). Er wird bei den meisten Befehlen vom Prozessor übersprungen. Wenn die Ausführung eines Befehls außer M1 weitere Maschinenzyklen beansprucht, d. h. M1, M2 oder mehr Zyklen, findet ein direkter Übergang von T4 in M1 nach T1 in M2 statt. Wir wollen ein Beispiel untersuchen. Die genaue interne Abfolge für jedes Beispiel ist in der Tabelle Abb. 2.27 gezeigt. Da diese Tabellen für den Z80 nicht veröffentlicht wurden, werden statt dessen die Tabellen vom 8080 verwendet. Sie liefern ein oberflächliches Verständnis für die Befehlsausführung.

LD D,C

Dies entspricht dem Befehl MOV r1,r2 beim 8080. Beachten Sie Zeile 1 von Abb. 2.27. Auch in diesem Beispiel wurde als Zielregister das Register D gewählt. Der Transfer ist in Abb. 2.20 abgebildet.

Dieser Befehl wurde im vorhergehenden Abschnitt beschrieben. Er überträgt den Inhalt von Register C, mit „C" bezeichnet, ins Register D. Die ersten drei Zustände vom Zyklus M1 werden verwendet, um den Befehl aus dem Speicher zu holen. Am Ende von T3 steht der Befehl in IR, dem Befehlsregister, von wo aus er dekodiert werden kann (siehe Abb. 2.19).

Während T4: (S S S) ▶ TMP

Der Inhalt von C wird ins TMP abgelegt (siehe Abb. 2.21).

Während T5: (TMP) ▶ DDD

Der Inhalt von TMP wird nach D übertragen. Dies ist in Abb. 2.22 gezeigt.

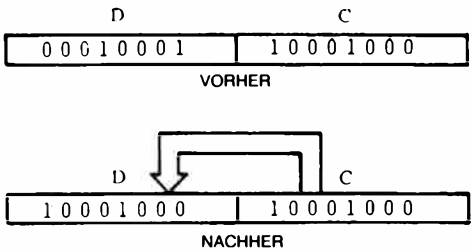


Abb. 2.20: Übertragung von C nach D

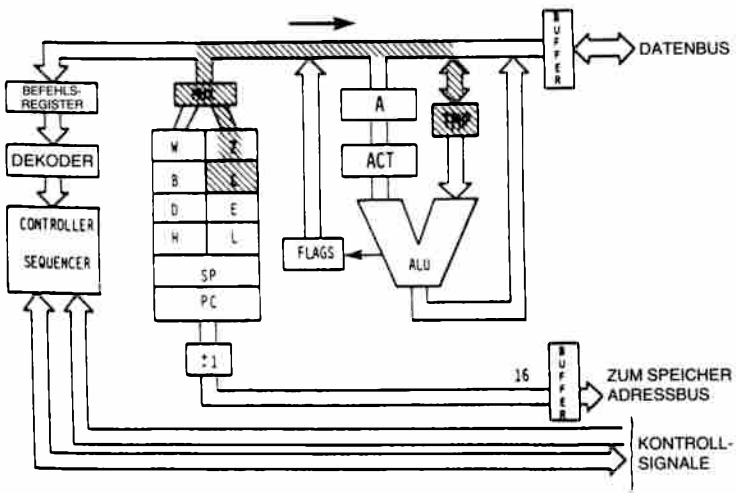


Abb. 2.21: Der Inhalt von C wird ins TMP übertragen

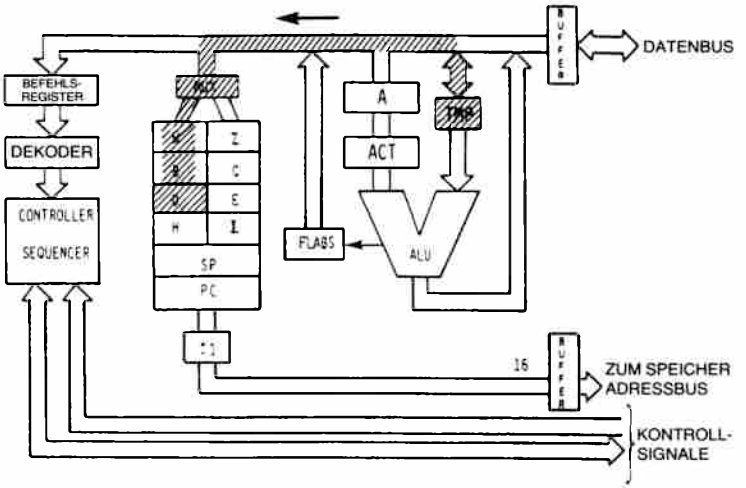


Abb. 2.22: Der Inhalt von TMP wird nach D übertragen

Der Befehl ist jetzt vollständig ausgeführt. Der Inhalt des Registers C wurde in das angegebene Zielregister D übertragen. Damit wird die Ausführung des Befehls beendet. Die anderen Maschinenzyklen M2, M3, M4 und M5 sind nicht notwendig und die Ausführung endet mit M1.

Die Ausführungszeit dieses Befehls kann leicht berechnet werden. Die Dauer jedes Zustands beim Standard-Z80 ist die Taktdauer: 500 ns. Die Ausführungszeit dieses Befehls ist die Dauer von fünf Zuständen oder 5 x 500 ns = 2500 ns = 2,5 μs.

Frage: Warum benötigt dieser Befehl zwei Zustände, T4 und T5, um den Inhalt von C nach D zu übertragen, und nicht nur einen Zustand? Er überträgt den Inhalt von C nach TMP und anschließend den Inhalt von TMP nach D. Wäre es nicht einfacher, den Inhalt von C innerhalb eines Zustands nach D zu übertragen?

Antwort: Wegen der Ausführung der internen Register ist dies nicht möglich. In Wirklichkeit sind alle internen Register Teil eines einzigen RAM, eines Schreib-/Lesespeichers innerhalb des Mikroprozessorbausteins. Zu einem Zeitpunkt kann nur ein Wort innerhalb des RAM adressiert oder ausgewählt werden (ein einziger Eingang). Aus diesem Grund ist es nicht möglich, zwei verschiedene Speicherplätze eines RAM gleichzeitig zu lesen und zu schreiben. Es sind daher zwei RAM-Zyklen nötig. Weiterhin ist es notwendig, die Daten zuerst aus dem Register-RAM zu lesen und in einem Zwischenspeicher, dem Register TMP, abzulegen, und sie dann in das endgültige Zielregister, hier das Register D, zurückzuschreiben. Dies ist eine Unzulänglichkeit des Konzepts. Allerdings ist diese Einschränkung bei nahezu allen monolithi-

schen Mikroprozessoren üblich. Um dieses Problem zu lösen, wäre ein RAM mit zwei Zugängen nötig. Dies ist keine prinzipielle Einschränkung bei Mikroprozessoren, und sie besteht bei Bit-Slice-Bausteinen normalerweise nicht. Sie ist ein Ergebnis der Bemühungen um die Pakkungsdichte auf dem Chip und mag in Zukunft behoben werden.

Wichtige Aufgabe:

An diesem Punkt sei es dem Leser dringend empfohlen, den Ablauf dieses einfachen Befehls selbst nochmals durchzusehen, bevor wir zu komplizierteren Befehlen fortschreiten. Gehen Sie zu diesem Zweck zu Abb. 2.14 zurück. Stellen Sie einige kleine „Symbole" wie Streichhölzer, Büroklammern usw. zusammen. Bewegen Sie dann die Symbole auf Abb. 2.14, um den Fluß der Daten aus den Registern in den Bus zu simulieren. Legen Sie beispielsweise ein Symbol auf PC. T1 bewegt das Symbol aus PC über den Adreßbus zum Speicher. Fahren Sie in diesem Sinne mit der simulierten Befehlsausführung fort, bis Ihnen der Transfer über die Busse und zwischen den Registern vertraut ist. Dann sollten Sie in der Lage sein, fortzufahren.

Jetzt werden zunehmend komplexere Befehlsabläufe studiert:

ADD A,r

Dieser Befehl heißt: Addiere den Inhalt von Register r (durch den Binärkode S S S festgelegt) zum Akkumulator (A) und lege das Ergebnis im Akkumulator ab. Dies ist ein *impliziter* Befehl. Er wird implizit genannt, weil er sich nicht ausdrücklich auf ein zweites Register bezieht. Der Befehl referiert nur das Register r explizit. Er setzt stillschweigend voraus, daß das andere Register, das in den Befehl einbezogen ist, der Akkumulator ist.

Wenn der Akkumulator in so einem impliziten Befehl verwendet wird, wird er sowohl als Quelle als auch als Ziel verwendet. Als das Ergebnis der Addition werden Daten im Akkumulator abgelegt. Der Vorteil eines solchen impliziten Befehls ist es, daß der gesamte Opcode nur acht Bit lang ist. Er verlangt nur ein drei Bit langes Registerfeld zur Festlegung von r. Dies ist eine schnelle Art, eine Addition durchzuführen.

In dem System gibt es weitere implizite Befehle, die sich auf andere spezielle Register beziehen. Kompliziertere Beispiele solcher impliziter Befehle sind z. B. die Befehle PUSH und POP, die Information zwischen dem obersten Element des Stapels und dem Akkumulator übertragen und gleichzeitig den Stapelzeiger (SP) modifizieren, indem sie ihn dekrementieren bzw. inkrementieren. Sie verändern implizit das Register Sp.

Die Ausführung des Befehls ADD A,r wird jetzt im Detail untersucht. Dieser Befehl nimmt zwei Maschinenzyklen M1 und M2 in Anspruch. Wie üblich wird der Befehl während der drei ersten Zustände von M1 aus dem Speicher geholt und in IR abgelegt. Am Beginn von T4 ist er de-

kodiert und kann ausgeführt werden. Nun wird angenommen, daß das Register B zum Akkumulator addiert wird. Der Befehlskode ist dann: 1 0 0 0 0 0 0 0 (der Kode für das Register B ist 0 0 0). Der entsprechende 8080-Befehl heißt ADD r.

T4 (S S S) ▸ TMP, (A) ▸ ACT

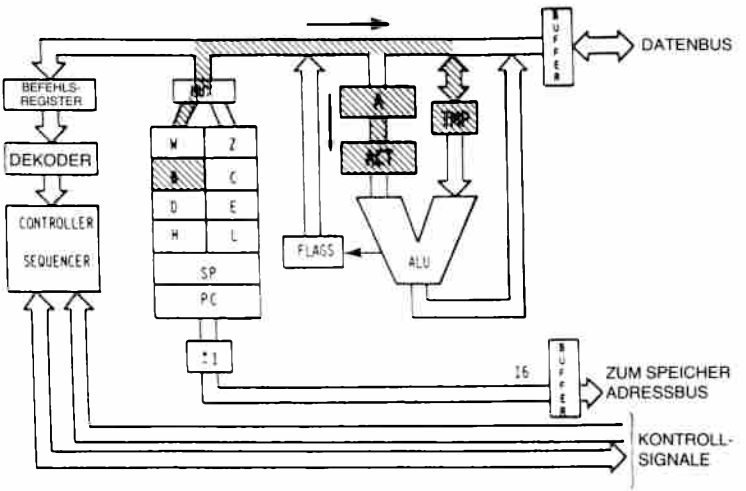


Abb. 2.23: Zwei Transfers werden gleichzeitig ausgeführt

Zwei Transfers werden gleichzeitig ausgeführt. Erstens wird der Inhalt des spezifizierten Quellregisters (hier B) nach TMP übertragen, d. h. zum rechten Eingang der ALU (siehe Abb. 2.23). Gleichzeitig wird der Inhalt des Akkumulators zum Akkumulator-Zwischenspeicher (ACT) übertragen. Wenn Sie Abb. 2.23 untersuchen, können Sie sich davon überzeugen, daß diese Transfers gleichzeitig stattfinden können. Sie verwenden getrennte Wege innerhalb des Systems. Der Transfer von B nach TMP benutzt den internen Datenbus. Der Transfer von A nach ACT verwendet einen kurzen internen Kanal, der von diesem Datenbus unabhängig ist. Um Zeit zu sparen, werden beide Transfers gleichzeitig ausgeführt. An diesem Punkt sind sowohl der rechte als auch der linke Eingang der ALU richtig geladen. Der linke Eingang ist vom Inhalt des Akkumulators bestimmt, der rechte Eingang vom Inhalt des Registers B. Wir sind nun zur Durchführung der Addition bereit. Normalerweise würden wir erwarten, daß die Addition während des Zustands T5 von M1 stattfindet. Dieser Zustand wird jedoch einfach nicht benutzt! Wir beginnen den Maschinenzyklus M2. Während des Zustands T1 passiert nichts! Erst im Zustand T2 von M2 wird die Addition ausgeführt (beachte ADD r in Abb. 2.27).

T2 von M2: (ACT) + (TMP) ▸ A

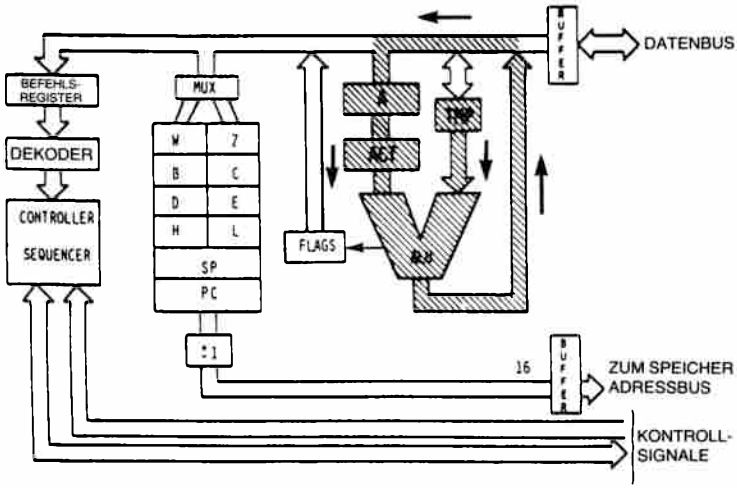


Abb. 2.24: Das Ende von ADD r

Der Inhalt von ACT wird zum Inhalt von TMP addiert und das Ergebnis schließlich im Akkumulator abgelegt. Siehe dazu Abb. 2.24. Die Operation ist jetzt vollständig ausgeführt.

Frage: Warum wurde die Ausführung der Addition bis zum Zustand T2 von Maschinenzyklus M2 verzögert, statt daß sie während des Zustands T5 von M1 stattfand? (Dies ist eine schwierige Frage, die Verständnis vom Aufbau einer CPU voraussetzt. Das benutzte Prinzip ist jedoch grundlegend für den Entwurf taktsynchronisierter CPUs. Wir wollen versuchen zu sehen, was passiert.)

Antwort: Wir verwenden einen üblichen „Trick", der in den meisten CPUs benutzt wird. Man nennt ihn Überlappung von Holen und Ausführung. Folgendes ist die zugrundeliegende Idee: Wenn man nochmals Abb. 2.23 betrachtet, kann man sehen, daß zur Ausführung der Addition nur die ALU und der Datenbus benötigt werden. Speziell wird nicht auf das Register-RAM (den Registerblock) zugegriffen. Wir (oder das Steuerwerk) wissen, daß die nächsten drei Zustände, die nach Abschluß eines Befehls ausgeführt werden, die Zustände T1, T2 und T3 vom Maschinenzyklus M1 des nächsten Befehls sein werden. Wenn wir nochmals die Ausführung dieser drei Zustände anschauen, dann sehen wir, daß dazu nur ein Zugriff auf den Befehlszähler (PC) und die Benutzung des Adreßbusses nötig ist. Zugriff auf den Befehlszähler heißt Zugriff auf das Register-RAM. (Dies erklärt, warum der gleiche Trick bei dem Befehl LD r,r' nicht verwendet werden konnte.) Es ist deshalb möglich, den schraffierten Bereich in Abb. 2.17 und den schraffierten Bereich in Abb. 2.24 gleichzeitig zu benutzen.

Der Datenbus wird während des Zustands T1 von M1 dazu benutzt, Statusinformationen auszuführen. Er kann nicht für die Addition verwendet werden, die wir durchführen wollen. Aus diesem Grund ist es notwendig, bis zum Zustand T2 zu warten, bevor die Addition wirklich ausgeführt werden kann. Entsprechend der Tabelle passiert folgendes: Die Addition wird während des Zustands T2 von M2 ausgeführt. Der Mechanismus ist jetzt erklärt. Der Vorteil dieser Methode sollte klar sein. Nehmen wir an, wir hätten ein einfaches Verfahren angewendet und die Addition während des Zustands T5 vom Maschinenzyklus M1 ausgeführt. Der Befehl ADD hätte dann 5 x 500 ns = 2500 ns gedauert. Mit dem Verfahren der Überlappung, das eingebaut wurde, wird der nächste Befehl begonnen, sobald der Zustand T4 ausgeführt ist. Auf eine Art und Weise, die für diesen nächsten Befehl nicht bemerkbar ist, verwendet die „kluge" Steuereinheit den Zustand T2, um das Ende der Addition auszuführen. In der Tabelle ist T2 als Teil von M2 gezeigt. Im Prinzip ist M2 der zweite Maschinenzyklus der Addition. In Wirklichkeit wird dieser M2 überlappt, d. h. er ist mit dem Maschinenzyklus M1 des nächsten Befehls identisch. Für den Programmierer ist die Verzögerung, die durch den Befehl ADD verursacht wird, nur vier Zustände, d. h. 4 x 500 ns = 2000 ns, anstatt 2500 ns, wenn man das „direkte" Verfahren benutzt. Es wurden 500 ns gespart, was einer Geschwindigkeitssteigerung von 20% entspricht.

Die Überlappungstechnik ist in Abb. 2.25 abgebildet. Sie wird immer verwendet, wenn es möglich ist, um die wirksame Ausführungsgeschwindigkeit des Mikroprozessors zu erhöhen. Natürlich ist eine Überlappung nicht immer möglich. Die nötigen Busse oder Einrichtungen müssen verfügbar sein, ohne daß ein Zugriffskonflikt auftritt. Das Steuerwerk „weiß", wann eine Überlappung möglich ist.

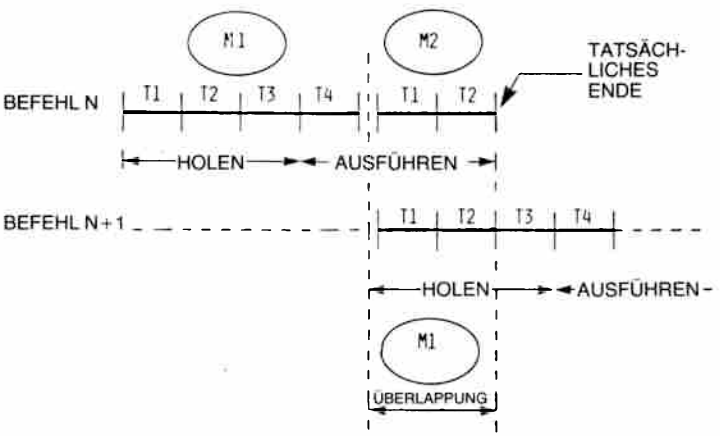


Abb. 2.25: Fetch-Execute-Überlappung während T1 – T2

Anmerkungen:

1. Der erste Speicherzyklus (M1) ist immer ein Fetch. Das erste (oder einzige) Byte, das den Opcode enthält, wird während dieses Zyklus geholt.

2. Ist der READY-Eingang vom Speicher während T2 jedes Speicherzyklus nicht high, fügt der Prozessor einen Wartezustand (TW) ein, bis READY high wird.

3. Wenn nötig, werden die Zustände T4 und T5 für Operationen benutzt, die vollständig innerhalb der CPU abgewickelt werden. Der Inhalt des internen Busses ist während T4 und T5 auf dem Datenbus verfügbar. Dies ist nur für Testzwecke vorgesehen. Ein „X" zeigt an, daß der Zustand anliegt, aber nur für interne Zwecke, wie zur Befehlsdekodierung benutzt wird.

4. Nur das Registerpaar rp=B (Register B und C) oder das Registerpaar rp=D (Register D oder E) kann ausgewählt werden.

5. Drei Zustände wurden übersprungen.

6. Speicher-Lesezyklus: Ein Befehl oder ein Datenwort wird gelesen.

7. Speicherzyklus.

8. Das READY-Signal wird während des zweiten und dritten Zyklus (M2 und M3) nicht benötigt. Das HOLD-Signal wird während M2 und M3 akzeptiert. Das SYNC-Signal wird während M2 und M3 nicht erzeugt. Während der Ausführung von DAD werden M2 und M3 zur internen Addition eines Registerpaares benutzt, der Speicher wird nicht angesprochen.

9. Das Ergebnis dieser arithmetischen, logischen oder Rotierbefehle wird nicht vor dem Zustand T2 des nächsten Befehlszyklus in den Akkumulator (A) übertragen. Das heißt, A wird geladen, während der nächste Befehl geholt wird. Dieses Überlappen von Befehlen erlaubt eine schnellere Verarbeitung.

10. Wenn der Wert der unteren vier Bit des Akkumulators größer als neun ist oder wenn das zusätzliche Übertragsbit gesetzt ist, wird 6 zum Akkumulator addiert. Ist der Wert der oberen vier Bit des Akkumulators dann größer als 9 oder ist das Übertragsbit gesetzt, dann wird 6 zu den oberen vier Bit des Akkumulators addiert.

11. Dies bedeutet den ersten Maschinenzyklus (den Fetch-Zyklus) des nächsten Befehlszyklus.

12. War die Bedingung erfüllt, wird der Inhalt des Registerpaares WZ statt dem Inhalt des Befehlszählers (PC) auf den Adreßbus (A_{0-15}) ausgegeben.

13. War die Bedingung nicht erfüllt, werden die Maschinenzyklen M4 und M5 übersprungen. Der Prozessor führt stattdessen sofort den Fetch-Zyklus (M1) des nächsten Befehlszyklus weiter.

14. War die Bedingung nicht erfüllt, werden die Maschinenzyklen M2 und M3 übersprungen. Der Prozessor führt stattdessen sofort den Fetch-Zyklus (M1) des nächsten Befehlszyklus aus.

15. Stack-Lese-Teil-Zyklus

16. Stack-Schreib-Teil-Zyklus

17.

Bedingung	CCC
NZ – nicht Null (Z=0)	000
Z – Null (Z=1)	001
NC – kein Übertrag (CY=0)	010
C – Übertrag (CY=1)	011
PO – ungerade Parität (P=0)	100
PE – gerade Parität (P=1)	101
P – plus (S=0)	110
M – Minus (S=1)	111

18. Ein-/Ausgabezyklus: Der 8-Bit-Kode des ausgewählten I/O-Ports wird gleichzeitig an die Adreßleitungen 0–7 (A_{0-7}) und 8–15 (A_{8-15}) ausgegeben.

19. Ausgabezyklus

20. Der Prozessor verbleibt untätig im Halt-Zustand, bis ein Interrupt, ein Reset oder ein Hold akzeptiert wird. Wird eine Hold-Anfrage akzeptiert, geht die CPU in den Hold-Modus. Endet der Hold-Modus, kehrt die CPU zum Halt-Zustand zurück. nach einem Reset beginnt der Prozessor die Ausführung bei der Adresse Null. Nach einem Interrupt führt der Prozessor den Befehl aus, der auf dem Datenbus erscheint (üblicherweise ein Restart).

SSS oder DDD	Wert
A	111
B	000
C	001
D	010
E	011
H	100
L	101

rp	Wert
B	00
D	01
H	10
SP	11

Abb. 2.26: Abkürzungen von INTEL

Frage: Wäre es möglich, weiter nach diesem Schema vorzugehen und auch den Zustand T3 von M2 zu verwenden, wenn wir einen längeren Befehl ausführen müssen?

Um den internen Mechanismus des Ablaufs zu klären, empfehle ich, die Abb. 2.27 zu untersuchen, die die Befehlsausführung beim 8080 genau zeigt. Der Z80 enthält alle 8080 Befehle und mehr. Die Information, die in Abb. 2.27 dargestellt ist, ist für den Z80 nicht verfügbar. Die Abbildung wurde hier aufgenommen, um die internen Operationen dieses Mikroprozessors verständlich zu machen. Die Äquivalenz zwischen den Befehlen von Z80 und 8080 ist im Anhang F und G aufgelistet.

Jetzt wollen wir einen komplizierteren Befehl untersuchen:

ADD A,(HL)

Der Opcode für diesen Befehl ist 10000110. Dieser Befehl bedeutet: Addiere zum Akkumulator den Inhalt der Speicherstelle (HL). Die Speicherstelle wird durch ein ziemlich merkwürdiges Verfahren bestimmt. Es ist die Speicherstelle, deren Adresse in den Speicherzellen H und L enthalten ist. Dieser Befehl setzt voraus, daß diese beiden Spezialregister (HL) vor der Ausführung des Befehls geladen wurden. Der 16-Bit-Inhalt dieser Register legt die Adresse im Speicher fest, die die Daten enthält. Diese Daten werden zum Akkumulator addiert, und das Ergebnis bleibt im Akkumulator stehen.

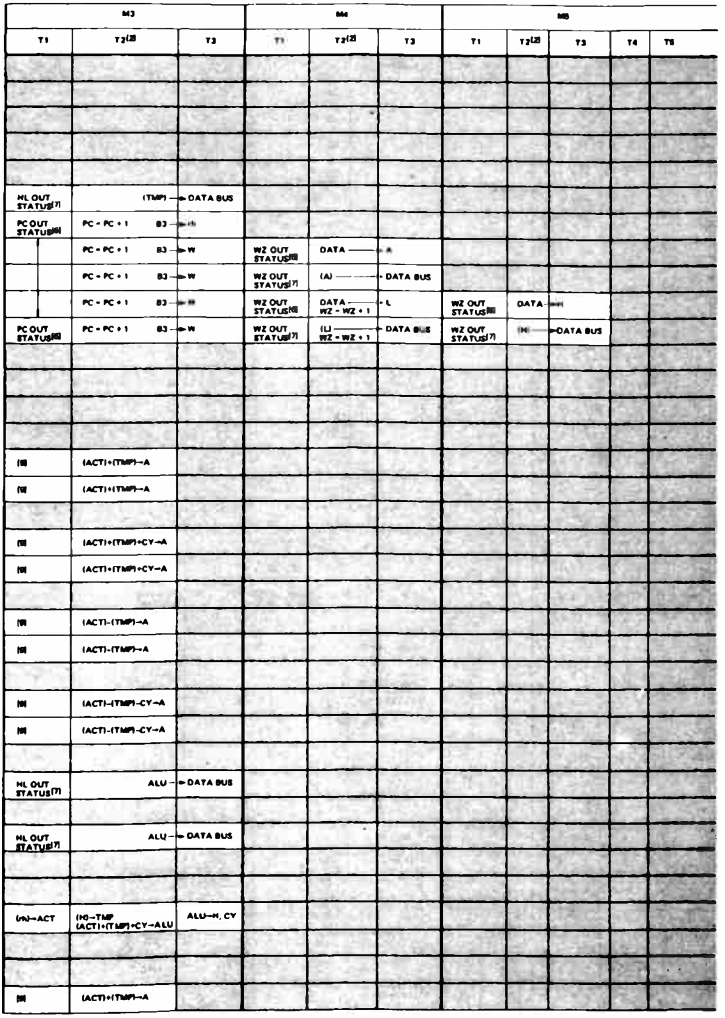
Dieser Befehl hat Geschichte. Er wurde eingebaut,um den 8080 kompatibel zu seinem Vorgänger, dem 8008 zu machen. Der alte 8008 hatte nicht die Möglichkeit, Speicher direkt zu adressieren! Um auf einen Speicherinhalt zuzugreifen, mußte man die beiden Register H und L laden und dann einen Befehl ausführen, der sich auf H und L bezog. ADD A,(HL) ist ein solcher Befehl. Es muß herausgestellt werden, daß der 8080 und der Z80 nicht in der gleichen Art wie der 8008 hinsichtlich der Speicheradressierung beschränkt sind. Sie haben die Möglichkeit, den Speicher direkt zu adressieren. Das Verfahren, die Register H und L zu verwenden, bietet einen zusätzlichen Vorteil und keine Einschränkung wie beim 8008.

Wir wollen nun die Ausführung dieses Befehls verfolgen (beim 8080 heißt er ADD M und ist der 16te Befehl in Abb. 2.27). Die Zustände T1, T2 und T3 von M1 werden wie üblich verwendet, um den Befehl zu holen. Während des Zustands T4 wird der Inhalt des Akkumulators zu seinem Pufferregister ACT übertragen, und der linke Eingang der ALU ist festgelegt.

Um das zweite Datenbyte zu erhalten, das zum Akkumulatur addiert werden soll, muß ein Speicherzugriff ausgeführt werden. Die Adresse dieses Datenbytes steht in H und L. Der Inhalt von H und L muß deshalb auf den Adreßbus übertragen werden, wo er zum Speicher geführt wird. Wir wollen dies durchführen.

MNEMONIC	OP CODE		M1[1]					M2		
	$D_7D_6D_5D_4$	$D_3D_2D_1D_0$	T1	T2[2]	T3	T4	T5	T1	T2[2]	T3
MOV r1, r2	0 1 D D	D S S S	PC OUT STATUS	PC = PC + 1	INST→TMP/IR	(SSS)→TMP	(TMP)→DDD			
MOV r, M	0 1 D D	D 1 1 0				x[3]		ML OUT STATUS[6]	DATA →	►DDD
MOV M, r	0 1 1 1	0 S S S				(SSS)→TMP		ML OUT STATUS[7]	(TMP) →	►DATA BUS
SPHL	1 1 1 1	1 0 0 1				(HL) →	SP			
MVI r, data	0 0 D D	D 1 1 0				x		PC OUT STATUS[6]	B2 →	►DDDD
MVI M, data	0 0 1 1	0 1 1 0				x			B2 →	►TMP
LXI rp, data	0 0 R P	0 0 0 1				x			PC = PC + 1 B2 →	►r1
LDA addr	0 0 1 1	1 0 1 0				x			PC = PC + 1 B2 →	►Z
STA addr	0 0 1 1	0 0 1 0				x			PC = PC + 1 B2 →	►Z
LHLD addr	0 0 1 0	1 0 1 0				x			PC = PC + 1 B2 →	►Z
SHLD addr	0 0 1 0	0 0 1 0				x		PC OUT STATUS[6]	PC = PC + 1 B2 →	►Z
LDAX rp[4]	0 0 R P	1 0 1 0				x		rp OUT STATUS[6]	DATA →	►A
STAX rp[4]	0 0 R P	0 0 1 0				x		rp OUT STATUS[7]	(A) →	►DATA BUS
XCHG	1 1 1 0	1 0 1 1				(HL)←→(DE)				
ADD r	1 0 0 0	0 S S S				(SSS)→TMP (A)→ACT		[9]	(ACT)+(TMP)→A	
ADD M	1 0 0 0	0 1 1 0				(A)→ACT		ML OUT STATUS[6]	DATA →	►TMP
ADI data	1 1 0 0	0 1 1 0				(A)→ACT		PC OUT STATUS[6]	PC = PC + 1 B2 →	►TMP
ADC r	1 0 0 0	1 S S S				(SSS)→TMP (A)→ACT		[9]	(ACT)+(TMP)+CY→A	
ADC M	1 0 0 0	1 1 1 0				(A)→ACT		ML OUT STATUS[6]	DATA →	►TMP
ACI data	1 1 0 0	1 1 1 0				(A)→ACT		PC OUT STATUS[6]	PC = PC + 1 B2 →	►TMP
SUB r	1 0 0 1	0 S S S				(SSS)→TMP (A)→ACT		[9]	(ACT)-(TMP)→A	
SUB M	1 0 0 1	0 1 1 0				(A)→ACT		ML OUT STATUS[6]	DATA →	►TMP
SUI data	1 1 0 1	0 1 1 0				(A)→ACT		PC OUT STATUS[6]	PC = PC + 1 B2 →	►TMP
SBB r	1 0 0 1	1 S S S				(SSS)→TMP (A)→ACT		[9]	(ACT)-(TMP)-CY→A	
SBB M	1 0 0 1	1 1 1 0				(A)→ACT		ML OUT STATUS[6]	DATA →	►TMP
SBI data	1 1 0 1	1 1 1 0				(A)→ACT		PC OUT STATUS[6]	PC = PC + 1 B2 →	►TMP
INR r	0 0 0 D	D 1 0 0				(DDD)→TMP (TMP) + 1→ALU	ALU→DDD			
INR M	0 0 1 1	0 1 0 0				x		ML OUT STATUS[6]	DATA → (TMP)+1 →	►TMP ►ALU
DCR r	0 0 0 D	D 1 0 1				(DDD)→TMP (TMP)+1→ALU	ALU→DDD			
DCR M	0 0 1 1	0 1 0 1				x		ML OUT STATUS[6]	DATA → (TMP)-1 →	►TMP ►ALU
INX rp	0 0 R P	0 0 1 1				(RP) + 1 →	RP			
DCX rp	0 0 R P	1 0 1 1				(RP) - 1 →	RP			
DAD rp[8]	0 0 R P	1 0 0 1				x		rp→ACT	(L)→TMP (ACT)+(TMP)→ALU	ALU→L, CY
DAA	0 0 1 0	0 1 1 1				DAA→A, FLAGS[10]				
ANA r	1 0 1 0	0 S S S				(SSS)→TMP (A)→ACT		[9]	(ACT)+(TMP)→A	
ANA M	1 0 1 0	0 1 1 0	PC OUT STATUS	PC = PC + 1	INST→TMP/IR	(A)→ACT		ML OUT STATUS[6]	DATA →	►TMP

Abb. 2.27: Intel Befehlsformate



© Intel. Reproduced by permission.

Abb. 2.27: Intel Befehlsformate (Fortsetzung)

MNEMONIC	OP CODE		M1[1]					M2		
	D_7 D_6 D_5 D_4	D_3 D_2 D_1 D_0	T1	T2[2]	T3	T4	T5	T1	T2[2]	T3
ANI data	1 1 1 0	0 1 1 0	PC OUT STATUS	PC ← PC + 1	INST→TMP/IR	(A)→ACT		PC OUT STATUS[6]	PC ← PC + 1 B2 →	TMP
XRA r	1 0 1 0	1 S S S				(A)→ACT (SSS)→TMP		[9]	(ACT)+(TPM)→A	
XRA M	1 0 1 0	1 1 1 0				(A)→ACT		HL OUT STATUS[6]	DATA →	TMP
XRI data	1 1 1 0	1 1 1 0				(A)→ACT		PC OUT STATUS[6]	PC ← PC + 1 B2 →	TMP
ORA r	1 0 1 1	0 S S S				(A)→ACT (SSS)→TMP		[9]	(ACT)+(TMP)→A	
ORA M	1 0 1 1	0 1 1 0				(A)→ACT		HL OUT STATUS[6]	DATA →	TMP
ORI data	1 1 1 1	0 1 1 0				(A)→ACT		PC OUT STATUS[6]	PC ← PC + 1 B2 →	TMP
CMP r	1 0 1 1	1 S S S				(A)→ACT (SSS)→TMP		[9]	(ACT)-(TMP), FLAGS	
CMP M	1 0 1 1	1 1 1 0				(A)→ACT		HL OUT STATUS[6]	DATA →	TMP
CPI data	1 1 1 1	1 1 1 0				(A)→ACT		PC OUT STATUS[6]	PC ← PC + 1 B2 →	TMP
RLC	0 0 0 0	0 1 1 1				(A)→ALU ROTATE		[9]	ALU→A, CY	
RRC	0 0 0 0	1 1 1 1				(A)→ALU ROTATE		[9]	ALU→A, CY	
RAL	0 0 0 1	0 1 1 1				(A), CY→ALU ROTATE		[9]	ALU→A, CY	
RAR	0 0 0 1	1 1 1 1				(A), CY→ALU ROTATE		[9]	ALU→A, CY	
CMA	0 0 1 0	1 1 1 1				$(\bar{A})$→A				
CMC	0 0 1 1	1 1 1 1				$\overline{CY}$→CY				
STC	0 0 1 1	0 1 1 1				1→CY				
JMP addr	1 1 0 0	0 0 1 1				X		PC OUT STATUS[6]	PC ← PC + 1 B2 →	Z
J cond addr[17]	1 1 C C	C 0 1 0				JUDGE CONDITION		PC OUT STATUS[6]	PC ← PC + 1 B2 →	Z
CALL addr	1 1 0 0	1 1 0 1				SP ← SP - 1		PC OUT STATUS[6]	PC ← PC + 1 B2 →	Z
C cond addr[17]	1 1 C C	C 1 0 0				JUDGE CONDITION IF TRUE, SP ← SP - 1		PC OUT STATUS[6]	PC ← PC + 1 B2 →	Z
RET	1 1 0 0	1 0 0 1				X		SP OUT STATUS[15]	SP ← SP + 1 DATA →	Z
R cond addr[17]	1 1 C C	C 0 0 0			INST→TMP/IR	JUDGE CONDITION[16]		SP OUT STATUS[15]	SP ← SP + 1 DATA →	Z
RST n	1 1 N N	N 1 1 1			PC → W INST→TMP/IR	SP ← SP - 1		SP OUT STATUS[16]	SP ← SP - 1 (PCH) →	DATA BUS
PCHL	1 1 1 0	1 0 0 1			INST→TMP/IR	(HL) → PC				
PUSH rp	1 1 R P	0 1 0 1				SP ← SP - 1		SP OUT STATUS[16]	SP ← SP - 1 (rh) →	DATA BUS
PUSH PSW	1 1 1 1	0 1 0 1				SP ← SP - 1		SP OUT STATUS[16]	SP ← SP - 1 (A) →	DATA BUS
POP rp	1 1 R P	0 0 0 1				X		SP OUT STATUS[15]	SP ← SP + 1 DATA →	r1
POP PSW	1 1 1 1	0 0 0 1				X		SP OUT STATUS[15]	SP ← SP + 1 DATA →	FLAGS
XTHL	1 1 1 0	0 0 1 1				X		SP OUT STATUS[15]	SP ← SP + 1 DATA →	Z
IN port	1 1 0 1	1 0 1 1				X		PC OUT STATUS[6]	PC ← PC + 1 B2 →	Z, W
OUT port	1 1 0 1	0 0 1 1				X		PC OUT STATUS[6]	PC ← PC + 1 B2 →	Z, W
EI	1 1 1 1	1 0 1 1				SET INTE F/F				
DI	1 1 1 1	0 0 1 1				RESET INTE F/F				
HLT	0 1 1 1	0 1 1 0				X		PC OUT STATUS	HALT MODE[20]	
NOP	0 0 0 0	0 0 0 0	PC OUT STATUS	PC ← PC + 1	INST→TMP/IR	X				

Abb. 2.27: Intel Befehlsformate (Fortsetzung)

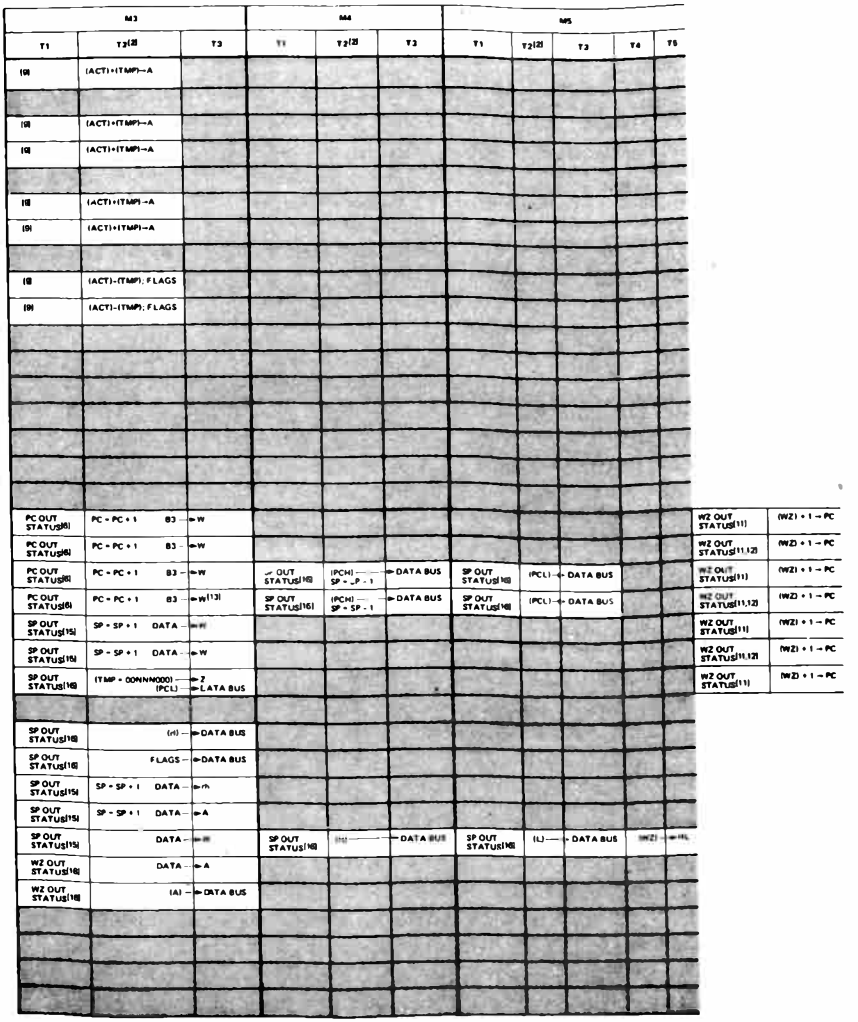


Abb. 2.27: Intel Befehlsformate (Fortsetzung)

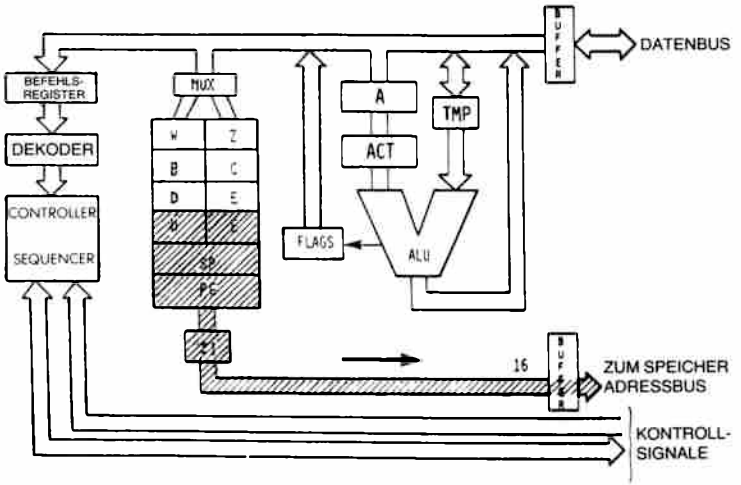


Abb. 2.28: Übertragung des Inhalts von HL zum Adreßbus

Während des Maschinenzyklus M2 lesen wir: HLOUT. H und L sind auf die gleiche Art auf den Adreßbus ausgegeben worden, auf die in den vorhergehenden Befehlen PC ausgegeben wurde. Es sei angemerkt, daß während des Zustands T1 der *Status* auf dem Datenbus ausgegeben wird (darauf wurde schon hingewiesen). Davon wird hier aber kein Gebrauch gemacht. Vereinfacht betrachtet braucht man zwei Zustände: einen, in dem der Speicher die Daten bereitstellt, und einen zweiten, in dem die Daten verfügbar sind und zum rechten Eingang der ALU (ins TMP) übertragen werden.

Beide Eingänge der ALU sind jetzt festgelegt. Die Situation ist analog zu der, die wir im vorhergehenden Befehl ADD A,r hatten: An beiden Eingängen der ALU stehen gültige Daten. Wir müssen einfach wie vorher addieren. Eine Fetch-Execute-Überlappungstechnik wird angewandt, und statt die Addition während des Zustands T4 von M2 auszuführen, wird sie zum Zustand T2 von M3 verschoben. In Abb. 2.27 sieht man, daß während T2 in der Tat ausgeführt wird: ACT + TMP ▶ A. Die Addition ist schließlich ausgeführt, die Inhalte von ACT und TMP sind addiert, und das Ergebnis ist im Akkumulator abgelegt.

Frage: Wie lange ist die scheinbare Ausführungszeit dieses Befehls (für den Programmierer)? Ist sie 3,6 µs oder 2,8 µs, wenn die Taktfrequenz 2,5 MHz beträgt?

Jetzt werden wir einen anderen komplexeren Befehl untersuchen, der eine direkte Speicheradressierung verwendet, wobei er zwei nicht zugängliche Register W und Z benutzt:

LD A,(nn)

Der Opcode ist 00111010. Das Äquivalent beim 8080 heißt LDA addr. Wie üblich werden die Zustände T1, T2 und T3 von M1 verwendet, um den Befehl aus dem Speicher zu holen. T4 wird ebenfalls benutzt, aber man kann keinen sichtbaren Vorgang beobachten. Tatsächlich wird der Befehl während des Zustands T4 dekodiert. Das Steuerwerk findet dabei heraus, daß es die beiden nächsten Bytes dieses Befehls holen muß, um die Adresse zu erhalten, deren Inhalt in dem Akkumulator geladen wird. Die Wirkung dieses Befehls ist es, daß der Akkumulator aus der Speicherzelle geladen wird, deren Adresse in den Bytes 2 und 3 des Befehls festgelegt ist. Beachten Sie, daß der Zustand T4 notwendig ist, um den Befehl zu dekodieren. Man könnte dies als Zeitverschwendung ansehen, da nur ein Teil dieses Zustands benötigt wird, um die Dekodierung durchzuführen. Dies ist auch so. Es ist allerdings der Grundgedanke der *getakteten Logik*. Weil intern *Mikrobefehle* verwendet werden, um die Dekodierung und Ausführung durchzuführen, muß man diesen Nachteil für den Vorzug der Mikroprogrammierung in Kauf nehmen. Die Struktur des Befehls erscheint in Abb. 2.29.

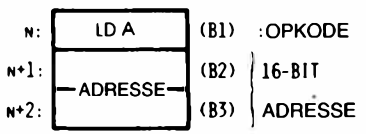


Abb. 2.29: LD A,(Adresse) ist ein Dreiwortbefehl

Jetzt werden die nächsten beiden Byte geholt. Sie legen eine Adresse fest (siehe Abb. 2.30).

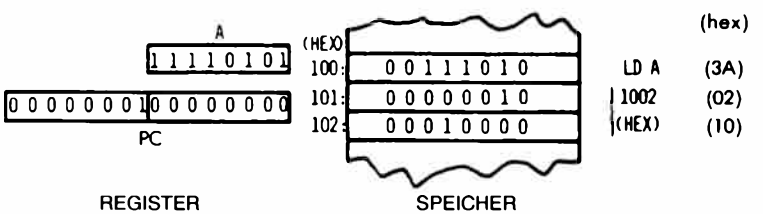


Abb. 2.30: Vor der Ausführung von LD A

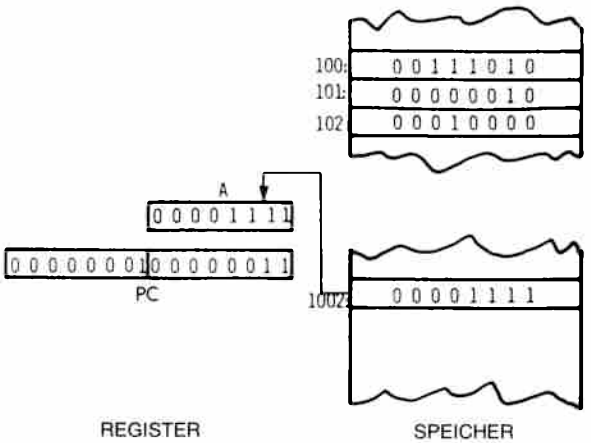


Abb. 2.31: Nach der Ausführung von LD A

Die Wirkung des Befehls ist in den Abb. 2.30 und 2.31 oben dargestellt.

Für das Steuerwerk sind zwei spezielle Register innerhalb des Z80 verfügbar (aber nicht für den Programmierer). Sie heißen „W" und „Z" und sind in Abb. 2.28 dargestellt.

Der zweite Maschinenzyklus M2: Wie üblich dienen die beiden ersten Zustände T1 und T2 dazu, den Inhalt der Speicherzelle PC zu holen. Während T2 wird der Befehlszähler PC inkrementiert. Gegen Ende von T2 werden die Daten aus dem Speicher verfügbar und erscheinen auf dem Datenbus. Mit dem Ende von T3 ist das Wort, das aus der Speicherzelle PC geholt wurde (B2, das zweite Byte des Befehls), auf dem Datenbus verfügbar. Es muß jetzt in einem Register zwischengespeichert werden. Es wird in Z abgelegt: B2 ▶ Z (siehe Abb. 2.32).

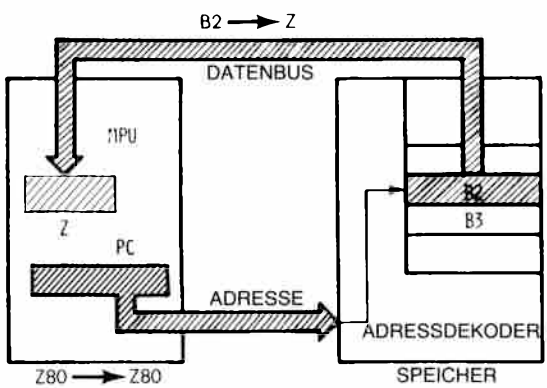


Abb. 2.32: Das zweite Byte des Befehls gelangt nach Z

Maschinenzyklus M3: Wieder wird PC auf dem Adreßbus ausgegeben, inkrementiert und schließlich wird das dritte Byte B3 aus dem Speicher gelesen und in das Register W des Mikroprozessors abgelegt. Zu diesem Zeitpunkt, d. h. am Ende des Zustands T3 von M3, enthalten die Register W und Z innerhalb des Mikroprozessors B2 und B3, d. h. die vollständige 16-Bit-Adresse, die ursprünglich in den beiden Worten stand, die dem Befehl im Speicher folgen. Die Ausführung kann jetzt vollendet werden. W und Z enthalten eine Adresse. Diese Adresse muß zum Speicher geschickt werden, um die richtigen Daten auszulesen. Dies geschieht im nächsten Speicherzyklus:

Maschinenzyklus M4: Jetzt werden W und Z auf den Adreßbus ausgegeben. Die 16-Bit-Adresse wird zum Speicher geschickt, und mit dem Ende von T2 stehen die Daten aus der angesprochenen Speicherzelle zur Verfügung. Sie werden schließlich am Ende des Zustands T3 in A abgelegt. Damit ist der Befehl vollständig ausgeführt.

Dies veranschaulicht die Verwendung eines *direkten Befehls.* Der Befehl belegt drei Bytes, um eine direkte Zwei-Byte-Adresse zu speichern. Der Befehl verwendet auch vier Speicherzyklen, dreimal mußte der Speicher angesprochen werden, um die drei Bytes des Befehls zu lesen. Ein zusätzlicher Speicherzugriff holte das Datenbyte, das durch die Adresse bestimmt war. Es ist ein langer Befehl. Dieser Befehl ist jedoch wichtig, wenn man den Akkumulator mit einem speziellen Inhalt laden will, der an einer bekannten Adresse im Speicher steht. Es sei angemerkt, daß dieser Befehl die Register W und Z benötigt.

Frage: Hätte dieser Befehl statt der Register W und Z auch andere Register innerhalb des Systems verwenden können?

Antwort: Nein. Hätte dieser Befehl andere Register verwendet, z. B. die Register H und L, dann hätte er deren Inhalt verändert. Nach der Ausführung des Befehls wäre der Inhalt der Register zerstört. Es wird in einem Programm immer vorausgesetzt, daß ein Befehl keine Register verändert, außer die Register, die er ausdrücklich anspricht. Ein Befehl, der den Akkumulator lädt, sollte den Inhalt keines anderen Registers zerstören. Aus diesem Grund ist es nötig, dem Steuerwerk die beiden zusätzlichen Register W und Z zum internen Gebrauch zur Verfügung zu stellen.

Frage: Wäre es möglich, den Befehlszähler PC statt der Register W und Z zu verwenden?

Antwort: Natürlich nicht. Das wäre Selbstmord. Der Leser sollte dies selbst untersuchen.

Jetzt werden wir einen weiteren Befehlstyp studieren: den Befehl *Verzweigung* oder *Sprung*, der dieReihenfolge ändert, in der Befehle innerhalb des Programms abgearbeitet werden. Bisher haben wir angenommen, daß Befehle aufeinanderfolgend ausgeführt werden. Es gibt aber

Befehle, die es dem Programmierer erlauben, aus der Folge heraus zu einem anderen Befehl innerhalb des Programms zu springen, oder praktisch gesagt, die zu einem anderen Speicherbereich oder einer anderen Adresse springen. Ein solcher Befehl ist:

JP nn

Dieser Befehl erscheint in Zeile 18 von Abb. 2.27 als „JMP addr". Seine Ausführung wird beschrieben, indem wir die Zeile in der Tabelle horizontal verfolgen. Dies ist wieder ein Dreiwortbefehl. Das erste Wort ist der Opcode und es enthält 11000011. Die nächsten beiden Worte enthalten die 16-Bit-Adresse, zu der der Sprung führt. Prinzipiell ist es die Wirkungsweise dieses Befehls, daß der Inhalt des Befehlszählers durch die 16 Bit, die dem „JUMP"-Opcode folgen, ersetzt wird. Praktisch wird aus Gründen der Effektivität ein etwas anderes Verfahren angewendet.

Wie zuvor wird in den ersten drei Zuständen von M1 der Befehl hereingeholt. Während des Zustands T4 wird er dekodiert und kein anderes Ereignis ist eingetragen (X). Die nächsten beiden Maschinenzyklen werden dazu verwendet, die Bytes B2 und B3 des Befehls zu holen. Während M2 wird B2 geholt und in W abgelegt. Die nächsten beiden Schritte werden von dem Prozessor während des nächsten Fetch ausgeführt, wie es auch bei der Addition der Fall war. Sie werden statt der üblichen Schritte während T1 und T2 des nächsten Befehls ausgeführt. Wir wollen sie anschauen.

Die beiden nächsten Schritte sind: WZ OUT und (WZ)+1 ▶ PC. Mit anderen Worten, während des nächsten Fetch-Zyklus wird der Inhalt von WZ statt dem Inhalt von PC ausgegeben. Das Steuerwerk hat registriert, daß ein Sprung ausgeführt wurde und wird den nächsten Befehl anders beginnen.

Die beiden zusätzlichen Zustände haben folgende Wirkung:

Die Adresse, die auf den Adreßbus des Systems ausgegeben wird, ist die Adresse, die in W und Z steht. Mit anderen Worten, der nächste Befehl wird von der Adresse geholt, die in W und Z steht. Dies bedeutet einen Sprung. Zusätzlich wird der Inhalt von WZ um eins inkrementiert und in den Befehlszähler abgelegt, so daß der nächste Befehl richtig geholt wird, indem PC wie üblich verwendet wird. Die Wirkungsweise ist also in Ordnung.

Frage: Warum haben wir den Inhalt von PC nicht direkt geladen? Warum verwenden wir W und Z zur Zwischenspeicherung?

Antwort: PC kann nicht verwendet werden. Hätten wir den unteren Teil von PC (PCL) mit B2 geladen, anstatt B2 zu verwenden, hätten wir den Inhalt von PC zerstört! Es wäre dann unmöglich gewesen, B3 zu laden.

Frage: Wäre es möglich statt W und Z alleine Z zu verwenden?

Antwort: Ja, das wäre möglich, aber es wäre langsamer. Wir hätten Z mit B2 laden, dann B3 hereinholen und in der oberen Hälfte von PC (PCH)

ablegen können. Dann müßten wir jedoch Z nach PCL übertragen, bevor wir den Inhalt von PC verwenden könnten. Dies würde den gesamten Ablauf verlangsamen. Aus diesem Grund sollte man sowohl W als auch Z verwenden. Weiterhin werden W und Z nicht nach PC übertragen, um Zeit zu sparen. Sie werden direkt zum Adreßbus ausgegeben, um den nächsten Befehl zu holen. Das Verständnis dieses Punktes ist entscheidend, wenn man die effiziente Ausführung von Befehlen innerhalb des Mikroprozessors verstehen will.

Frage: Nur für den interessierten und informierten Leser) Was passiert, wenn am Ende von M3 ein Interrupt auftritt?(Wenn die Befehlsausführung an diesem Punkt unterbrochen wird, zeigt der Befehlszähler auf den Befehl, der dem Sprungbefehl folgt, und die Sprungadresse, die in W und Z steht, geht verloren.)

Die Beantwortung bleibt als interessante Aufgabe dem interessierten Leser überlassen.

Die eingehenden Beschreibungen, die wir für die Ausführung typischer Befehle gegeben haben, sollte die Rolle der Register und der internen Busse klären. Um ein genaues Verständnis der internen Arbeitsweise des Z80 zu erhalten, mag ein zweites Durcharbeiten des vorhergehenden Abschnitts nützlich sein.

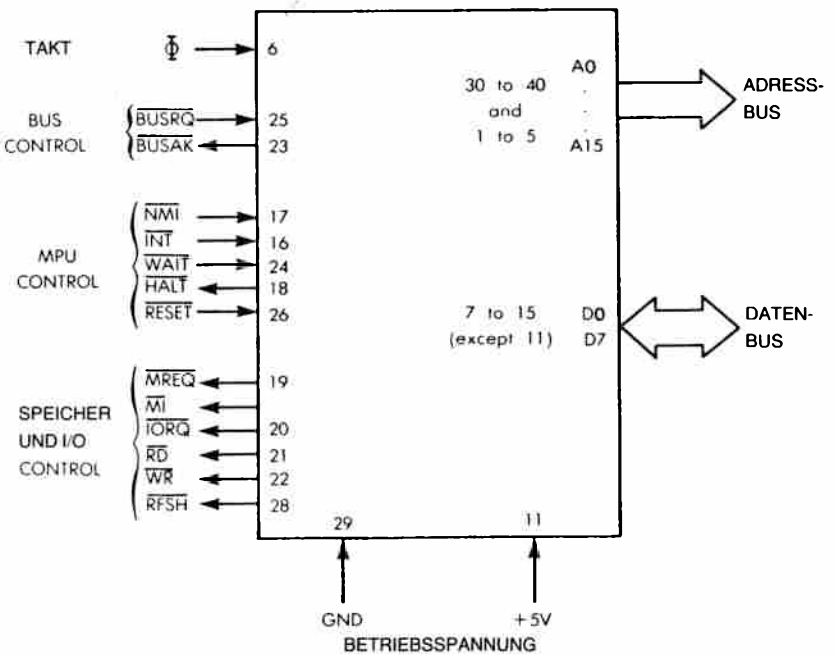


Abb. 2.33: Anschlußbelegung der Z80 MPU

Der Z80 Baustein

Der Vollständigkeit halber wollen wir hier die Signale des Z80 Mikroprozessorbausteins untersuchen. Es ist nicht unbedingt nötig, die Funktion der Z80 Signale zu verstehen, um den Z80 programmieren zu können. Der Leser, der an Einzelheiten der Hardware nicht interessiert ist, kann deshalb diesen Abschnitt überspringen. Die Anschlußbelegung des Z80 ist in Abb. 2.33 dargestellt. Auf der rechten Seite der Abbildung verrichten Daten- und Adreßbus ihre üblichen Aufgaben, wie sie am Anfang dieses Kapitels beschrieben wurden. Wir werden hier die Aufgabe der Signale im Kontrollbus beschreiben. Sie sind auf der linken Seite von Abb. 2.33 dargestellt.

Die Steuersignale wurden in vier Gruppen eingeteilt. Sie werden in der Reihenfolge der Abb. 2.33 von oben nach unten beschrieben.

Am Takteingang wird ein externer Taktgenerator angeschlossen, der den Systemtakt liefert.

Die beiden Steuersignale für den Bus, BUSRQ und BUSAK, dienen dazu, den Z80 von seinen Bussen abzukoppeln. Sie werden hauptsächlich von der DMA (Direct Memory Access) benutzt, können aber auch von einem anderen Prozessor im System verwendet werden. BUSRQ ist das Signal zur Anforderung der Busse. Es wird an den Z80 geschickt. Als Antwort bringt der Z80 seinen Adreßbus, seinen Datenbus und seine Tristate-Steuersignale am Ende des laufenden Maschinenzyklus in den hochohmigen Zustand. Mit dem Signal BUSAK bestätigt der Z80, daß seine Busausgänge hochohmig sind.

Sechs Z80-Steuersignale beziehen sich auf seinen internen Zustand oder seinen Ablauf:

INT und NMI sind die beiden Interruptsignale. INT ist die normale Interruptanforderung. Interrupts werden in Kapitel 6 beschrieben. Mehrere Ein-/Ausgabesteine können an die Interruptleitung INT angeschlossen werden. Wann immer eine Interruptanforderung auf dieser Leitung ansteht und das interne Interrupt-Flip-Flop (IFF) freigegeben ist, wird der Z80 einen Interrupt akzeptieren (vorausgesetzt BUSRQ ist nicht aktiv). Ein Rückmeldesignal wird ausgegeben: IORQ (während des M1-Zyklusses). Der Rest des Ablaufs wird in Kapitel 6 beschrieben.

NMI ist der nicht maskierbare Interrupt. Er wird vom Z80 jederzeit akzeptiert und erzwingt einen Sprung zur Adresse 0066 hexadezimal. Auch dies wird in Kapitel 6 beschrieben. (Hier wird ebenfalls vorausgesetzt, daß BUSRQ nicht aktiv ist.)

Das Signal WAIT dient dazu, den Z80 mit langsameren Speichern oder Ein-/Ausgabebausteinen zu synchronisieren. Ist es aktiv, so zeigt es an, daß der betreffende Speicher oder Baustein noch nicht zur Datenübertragung bereit ist. Die Z80 CPU geht dann in einen speziellen Wartezustand, bis das WAIT-Signal inaktiv wird. Dann wird der normale Ablauf fortgesetzt.

Mit dem Signal HALT zeigt der Z80 an, daß er den Befehl HALT ausgeführt hat. In diesem Zustand wartet der Z80 auf einen externen Interrupt und führt solange NOPs aus, um weiterhin den Speicher aufzufrischen.

Das Signal RESET initialisiert üblicherweise die MPU. Es setzt Befehlszähler, Register I und R auf „0". Es setzt das Interrupt-Flip-Flop zurück und setzt den Interrupt-Modus auf „0". Es wird normalerweise nach dem Einschalten der Betriebsspannung angelegt.

Steuerung von Speicher und Ein-/Ausgabe

Der Z80 erzeugt sechs Signale zur Steuerung von Speicher und Ein-/Ausgabebausteinen. Diese sind:

MREQ ist das Signal zur Anforderung des Speichers. Es zeigt an, daß auf dem Adreßbus eine gültige Adresse liegt. Dann kann eine Lese- oder Schreiboperation im Speicher ausgeführt werden.

M1 ist der Maschinenzyklus 1. Dies entspricht dem Fetch-Zyklus eines Befehls.

IORQ ist das Signal zur Anforderung eines Ein-/Ausgabebausteins. Es zeigt an, daß die Ein-/Ausgabeadresse auf den Bits 0–7 des Adressbusses gültig ist. Danach kann eine Lese-/Schreiboperation in dem Ein-/Ausgabebaustein ausgeführt werden. Ebenso wird IORQ zusammen mit M1 erzeugt, wenn der Z80 einen Interrupt zurückmeldet. Externe Bausteine können diese Signale verwenden, um den Interruptvektor auf den Datenbus zu legen. (Normale Ein-/Ausgabeoperationen finden nie während des M1-Zyklusses statt. Die Kombination von IORQ und M1 quittiert einen Interrupt.)

RD ist das Lesesignal. Es zeigt an, daß der Z80 bereit ist, den Inhalt des Datenbusses in den Akkumulator zu laden. Es kann von jedem externen Baustein, sei es ein Speicher oder ein Ein-/Ausgabebaustein, verwendet werden, um Daten auf den Datenbus zu legen.

WR ist das Schreibsignal. Es zeigt an, daß auf dem Bus gültige Daten liegen, die in den ausgewählten Baustein geschrieben werden können.

RFSH ist das Auffrischsignal. Ist RFSH aktiv, dann enthalten die unteren sieben Bit des Adreßbusses eine Auffrischadresse für dynamische Speicher. Das Signal MREQ wird dann zur Durchführung der Auffrischung verwendet, indem der Speicherinhalt gelesen wird.

Zusammenfassung

Damit ist unsere Beschreibung der internen Organisation des Z80 vollständig. Die genauen Einzelheiten der Hardware des Z80 sind hier nicht wichtig. Wichtig ist jedoch die Funktion jedes Registers, und dies sollte vollständig verstanden sein, bevor mit den nächsten Kapiteln weitergemacht wird. Jetzt werden die Befehle eingeführt, die wirklich auf dem Z80 verfügbar sind, und Grundtechniken der Programmierung vorgestellt.

3

Grundlegende Techniken der Programmierung

Einführung

Ziel dieses Kapitels ist es, die grundlegenden Techniken darzustellen, die man benötigt, um ein Programm für den Z80 zu schreiben. Dieses Kapitel wird neue Konzepte einführen, wie die Verwaltung von Registern, Schleifen und Unterprogramme. Es wird sich auf Programmiertechniken konzentrieren, die nur die *internen* Komponenten des Z80 verwenden, z. B. die Register. Es werden tatsächliche Programme wie Arithmetikprogramme entwickelt. Diese Programme werden die verschiedenen Konzepte veranschaulichen, die bis dahin vorgestellt sind, und sie verwenden reale Z80-Befehle. So wird gezeigt, wie man Befehle verwenden kann, um Information zwischen Speicher und der MPU zu übertragen und zu verarbeiten. Im nächsten Kapitel werden dann die Befehle, die auf dem Z80 verfügbar sind, bis in alle Einzelheiten diskutiert. Das Kapitel 5 wird die Adressierungstechniken vorstellen und das Kapitel 6 die Techniken, die man zur Verfügung hat, um Information außerhalb des Z80 zu verarbeiten: die Ein-/Ausgabetechniken.

In diesem Kapitel werden wir im wesentlichen durch Probieren lernen. Indem wir Programme von wachsender Kompliziertheit untersuchen, lernen wir die Aufgaben der verschiedenen Befehle und der Register kennen, und wir werden die bis dann vorgestellten Konzepte anwenden. Ein wichtiges Konzept wird hier aber noch nicht vorgestellt, die Adressierungstechniken. Diese sind nämlich so komplex, daß sie in einem eigenem Kapitel, dem Kapitel 5, behandelt werden.

Wir wollen nun damit beginnen, einige Programme für den Z80 zu schreiben. Wir fangen mit Arithmetikprogrammen an. Abbildung 3.0 zeigt den Z80 als „Modell für den Programmierer".

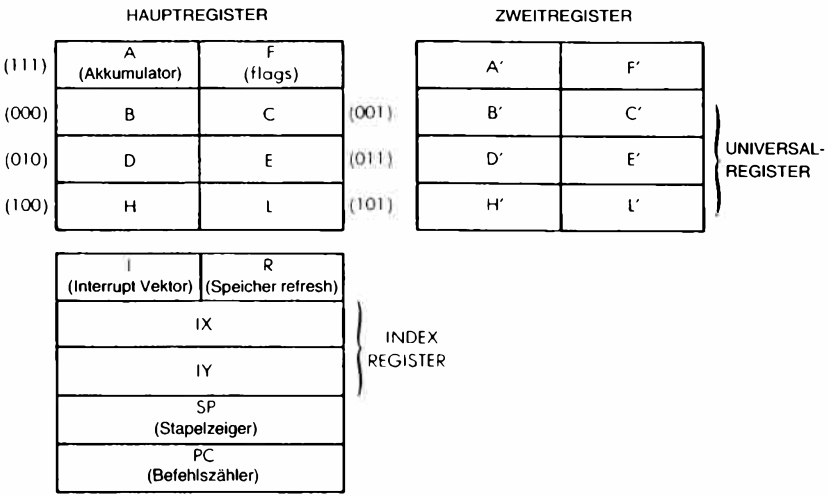


Abb. 3.0: Die Register des Z80

Arithmetikprogramme

Arithmetikprogramme sind Programme für Addition, Subtraktion, Multiplikation und Division. Die Programme, die hier vorgestellt werden, arbeiten mit ganzen Zahlen. Die ganzen Zahlen können dabei als positive Dualzahlen dargestellt werden, oder in Form von Zweierkomplementen, wobei das Bit ganz links das Vorzeichen angibt (siehe die Beschreibung des Zweierkomplements in Kapitel 1).

8-Bit-Addition

Wir wollen zwei Acht-Bit-Operanden addieren, die bei den Adressen ADR1 und ADR2 im Speicher stehen. Die Summe werde RES genannt und soll bei Adresse ADR3 im Speicher abgelegt werden. Dies ist in Abb. 3.1 veranschaulicht. Folgendes Programm führt die Addition durch:

Befehle	Kommentare
LD A,(ADR1)	Lade OP1 nach A
LD HL,ADR2	Lade HL mit der Adresse von OP2
ADD A,(HL)	Addiere OP2 zu OP1
LD (ADR3),A	Speichere Ergebnis nach ADR3

Dies ist unser erstes Programm. Die Befehle sind auf der linken Seite aufgelistet, rechts erscheinen Kommentare dazu. Wir wollen das Programm jetzt untersuchen. Es besteht aus vier Befehlen. Jede Zeile wird ein *Befehl* genannt, der hier in symbolischer Form dargestellt ist. Jeder solche Befehl wird von einem *Assembler-Programm* in ein, zwei, drei

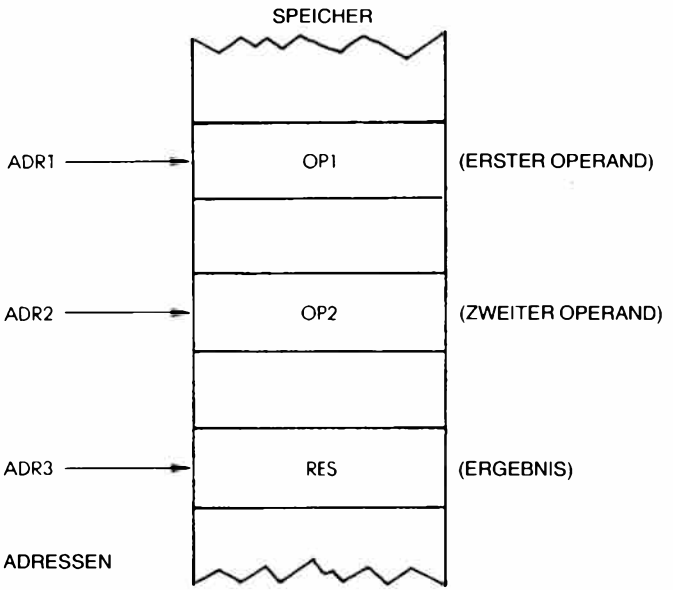


Abb. 3.1: Acht-Bit-Addition RES = OP1 + OP2)

oder vier Bytes übersetzt. Wir wollen uns hier nicht um die Übersetzung kümmern, sondern nur die symbolische Darstellung betrachten.

Die erste Zeile legt fest, daß der Inhalt von ADR1 in den Akkumulator geladen werden soll. Auf Abb. 3.1 bezogen ist der Inhalt von ADR1 der erste Operand „OP1". Dieser erste Befehl bewirkt, daß OP1 vom Speicher in den Akkumulator übertragen wird. Dies ist in Abb. 3.2 dargestellt. „ADR1" ist ein symbolischer Name für die wirkliche 16-Bit-Adresse im Speicher. Der Name ADR1 wird an einer anderen Stelle im Programm festgelegt. Beispielsweise könnte er definiert werden als die Adresse „100".

Dieser *Ladebefehl* führt zu einer Leseoperation aus Adresse 100 (siehe Abb. 3.2), deren Inhalt über den Datenbus übertragen und im Akkumulator abgelegt wird. Aus dem vorhergehenden Kapitel werden Sie sich noch daran erinnern, daß arithmetische und logische Operationen den Akkumulator als einen der Operanden verwenden. (Einzelheiten dazu im vorhergehenden Kapitel.) Weil wir die beiden Werte OP1 und OP2 addieren wollen, müssen wir zuerst OP1 in den Akkumulator laden. Danach können wir den Inhalt des Akkumulators, d. h. OP1 und OP2 addieren. Das Feld ganz rechts in diesem Befehl heißt *Kommentarfeld.* Bei der Übersetzung wird es von dem Assemblerprogramm nicht beachtet, aber es verbessert die Lesbarkeit des Programms. Um verstehen zu kön-

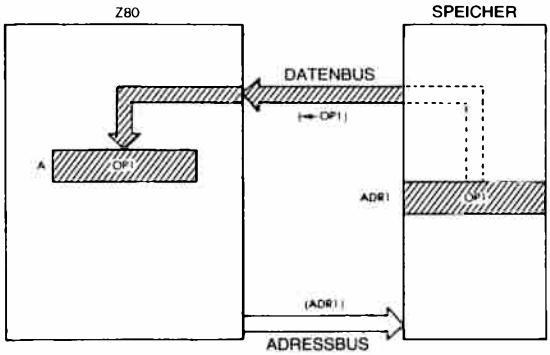


Abb. 3.2: LD A,(ADR1): OP1 wird aus dem Speicher geladen

nen, was das Programm tut, ist es von größter Wichtigkeit, gute Kommentare zu verwenden. Dies nennt man ein Programm dokumentieren.

Hier erklärt sich der Kommentar selbst: Der Wert von OP1, der bei der Adresse ADR1 steht, wird in den Akkumulator A geladen.

Die Wirkung dieses ersten Befehls wird in Abb. 3.2 veranschaulicht. Der zweite Befehl unseres Programms ist:

```
LD HL,ADR1
```

Dies heißt: „Lade ADR1 in die Register H und L." Um den zweiten Operanden OP2 aus dem Speicher zu lesen, müssen wir zuerst dessen Adresse in ein Registerpaar des Z80 laden, z. B. nach H und L. Dann können wir den Inhalt der Speicherzelle, deren Adrese in H und L steht, zum Akkumulator addieren.

```
ADD A,(HL)
```

Wenn wir uns auf Abb. 3.1 beziehen, dann ist der Inhalt der Speicherstelle ADR2 unser zweiter Operand OP2. Der Akkumulator enthält jetzt unseren ersten Operanden OP1. Die Ausführung dieses Befehls führt dazu, daß OP2 aus dem Speicher geholt und zu OP1 addiert wird. Dies ist in Abb. 3.3 dargestellt.

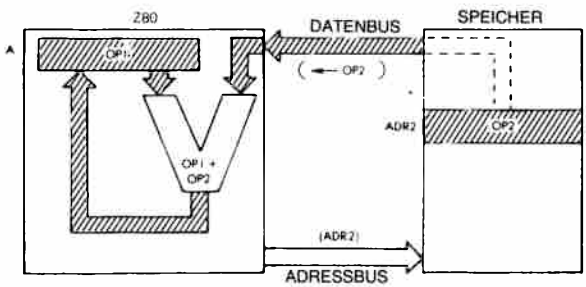


Abb. 3.3: ADD A,(HL)

Im Akkumulator steht dann die Summe. Der Leser wird sich daran erinnern, daß die Ergebnisse arithmetischer Operationen beim Z80 wieder in den Akkumulator abgelegt werden. Bei anderen Prozessoren kann es möglich sein, diese Ergebnisse in andere Register oder in den Speicher abzulegen.

Der Akkumulator enthält jetzt die Summe von OP1 und OP2. Um das Programm abzuschließen, müssen wir nur noch den Inhalt des Akkumulators in die Speicherstelle ADR3 übertragen, um das Ergebnis an der festgelegten Stelle abzuspeichern. Der vierte Befehl unseres Programms erledigt das:

LD (ADR3),A

Dieser Befehl lädt den Inhalt von A in die angegebene Adresse ADR3. Die Wirkungsweise dieses letzten Befehls ist in Abb. 3.4 dargestellt.

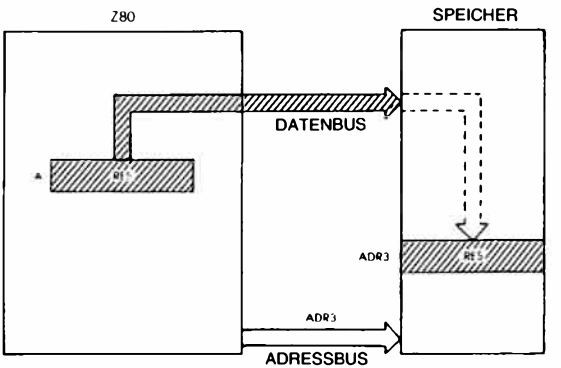


Abb. 3.4: LD (ADR3),A (Lege den Akkumulator im Speicher ab)

Vor der Ausführung des Befehls ADD enthielt der Akkumulator OP1 (siehe Abb. 3.3). Nach der Addition wurde ein neues Ergebnis in den Akkumulator geschrieben, nämlich „OP1 + OP2". Rufen Sie sich nochmals in Gedächtnis zurück, daß der Inhalt jedes Registers in dem Mikroprozessor wie auch der Inhalt jeder Speicherstelle unverändert bleibt, wenn eine Leseoperation auf dieses Register ausgeführt wurde. Mit anderen Worten: Wenn aus einem Register oder einer Speicherstelle gelesen wird, dann verändert sich der betreffende Inhalt nicht. Nur durch eine Schreiboperation in dieses Register wird sein Inhalt verändert. In unserem Beispiel bleibt der Inhalt der Speicherzellen ADR1 und ADR2 das ganze Programm hindurch unverändert. Nach dem Befehl ADD hat sich der Inhalt des Akkumulators jedoch geändert, weil das Ergebnis aus der ALU in den Akkumulator geschrieben wurde. Der alte Inhalt von A ist dann verloren.

Statt ADR1, ADR2 und ADR3 kann man konkrete Adressen verwenden. Will man aber die symbolischen Adressen beibehalten, muß man

sogenannte „Pseudobefehle" verwenden, die den Wert dieser symbolischen Adressen festlegen, so daß sie der Assembler bei der Übersetzung durch die wirklichen physikalischen Adressen ersetzen kann. Solche Pseudobefehle sind beispielsweise:

```
ADR1 = 100H      (H steht für hexadezimal)
ADR2 = 120H
ADR3 = 200H
```

Aufgabe 3.1: Machen Sie dieses Buch jetzt zu. Verwenden Sie nur die Befehlsliste am Ende des Buches. Schreiben Sie ein Programm, das zwei Zahlen addiert, die in den Speicherzellen LOC1 und LOC2 stehen. Legen Sie das Ergebnis in LOC3 ab. Vergleichen Sie Ihr Programm dann mit obigem Beispiel.

16-Bit-Addition

Eine Acht-Bit-Addition erlaubt nur die Addition von Acht-Bit-Zahlen, d. h. Zahlen zwischen 0 und 255, wenn man die absolute Dualdarstellung verwendet. Bei den meisten praktischen Anwendungen muß man aber Zahlen addieren, die 16 Bit lang oder länger sind, d. h. man muß *mehrfache Genauigkeit* verwenden. Wir wollen hier Beispiele für eine Arithmetik mit 16-Bit-Zahlen vorstellen. Diese können leicht auf 24, 32 oder mehr Bit erweitert werden (jeweils auf Vielfache von 8 Bit). Wir wollen annehmen, daß der erste Operand in den Speicherzellen ADR1

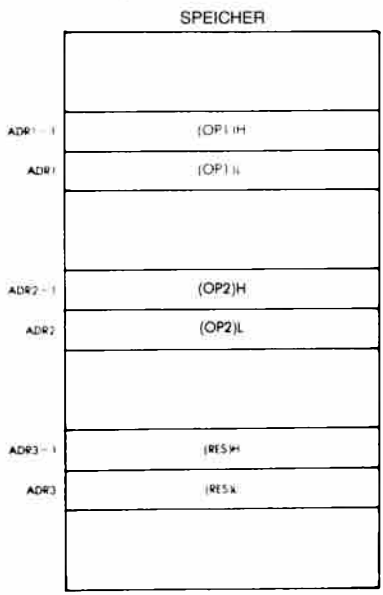


Abb. 3.5: 16-Bit-Addition – die Operanden

und ADR1-1 steht. Da OP1 jetzt eine 16-Bit-Zahl ist, nimmt er zwei 8-Bit-Speicherzellen ein. OP2 ist entsprechend bei ADR2 und ADR2-1 gespeichert. Das Ergebnis soll in den Speicherzellen ADR3 und ADR3-1 abgelegt werden. Dies ist in Abb. 3.5 dargestellt. H kennzeichnet die obere Hälfte (Bit 8 bis 15) und L die untere Hälfte (Bit 0 bis 7).

Das Prinzip dieses Programms ist genau das gleiche wie bei dem vorangehenden. Zuerst wird die untere Hälfte der beiden Operanden addiert, da der Mikroprozessor nur 8 Bit gleichzeitig addieren kann. Tritt bei der Addition dieser unteren Bytes ein Übertrag auf, so wird er automatisch im internen Übertragsbit („C") gespeichert. Dann werden die oberen Hälften der Operanden und ein eventueller Übertrag addiert, und das Ergebnis wird in den Speicher abgelegt. Das Programm lautet wie folgt:

```
LD   A,(ADR1)    Lade die untere Hälfte von OP1
LD   HL,ADR2     Adresse der unteren Hälfte von OP2
ADD  A,(HL)      Addiere die unteren Hälften von OP1 und OP2
LD   (ADR3),A    Speichere die untere Hälfte des Ergebnisses
LD   A,(ADR1-1)  Lade die obere Hälfte von OP1
DEC  HL          Adresse der oberen Hälfte von OP2
ADC  A,(HL)      (OP1 + OP2) oben + Übertrag
LD   (ADR3-1),A  Speichere die obere Hälfte des Ergebnisses
```

Die ersten vier Befehle dieses Programms sind die gleichen, die auch für die Acht-Bit-Addition im vorhergehenden Abschnitt verwendet wurden. Sie bewirken die Addition der unteren Hälften (Bit 0-7) von OP1 und OP2. Die Summe genannt „RES" wird in der Speicherstelle ADR3 abgelegt (siehe Abb. 3.5).

Immer wenn eine Addition durchgeführt wurde, wird der Übertrag (egal ob „0" oder „1") im Übertragsbit C des Flagregisters (Register F) gespeichert. Erzeugen die beiden Zahlen einen Übertrag, dann ist C gleich „1" (der Übertrag ist gesetzt). Erzeugen die beiden Zahlen keinen Übertrag, dann hat das Übertragsbit den Wert „0".

Die nächsten vier Befehle des Programms sind im wesentlichen die gleichen, die im vorhergehenden Programm zur Addition von 8-Bit-Zahlen verwendet wurden. Diesmal addieren sie die oberen Hälften (d. h. Bit 8-15) von OP1 und OP2 sowie einen eventuellen Übertrag und speichern das Ergebnis in Adresse ADR3-1.

Nach der Ausführung dieses Programms aus acht Befehlen, ist das 16 Bit lange Ergebnis in den Speicherzellen ADR3 und ADR3−1 abgelegt. Beachten Sie aber, daß zwischen der ersten und der zweiten Hälfte des Programms ein Unterschied besteht. Der Additionsbefehl, der verwendet wurde, ist nicht der gleiche wie in der ersten Hälfte. In der ersten Hälfte des Programms hatten wir den Befehl „ADD" verwendet (in der dritten Zeile). Dieser Befehl addiert die beiden Operanden, ohne das Übertragsbit zu beachten. In der zweiten Hälfte verwenden wir den Befehl „ADC", der die beiden Operanden addiert und zusätzlich einen Übertrag, der eventuell erzeugt worden war. Dies ist notwendig, um das

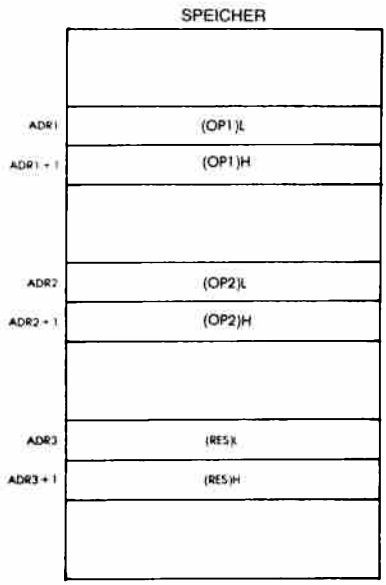


Abb. 3.6: Speicherung der Operanden in umgekehrter Reihenfolge

richtige Ergebnis zu erhalten. Wenn zuerst die unteren Hälften der Operanden addiert werden, kann ein Übertrag auftreten. Dieser eventuelle Übertrag muß bei der zweiten Hälfte der Addition berücksichtigt werden.

Jetzt ergibt sich natürlich folgende Frage: Was ist, wenn bei der Addition der oberen Hälften ebenfalls ein Übertrag auftritt? Es gibt zwei Möglichkeiten: Die erste ist, anzunehmen, daß dies ein Fehler ist. Dann ist das Programm dafür ausgelegt, Ergebnisse von bis zu 16 Bit, nicht aber 17 Bit zu verarbeiten. Die andere Möglichkeit ist es, zusätzliche Befehle anzuhängen, um ausdrücklich einen eventuellen Übertrag am Ende dieses Programms zu testen. Diese Entscheidung muß der Programmierer treffen, als erste von vielen Entscheidungen.

Beachten Sie: Wir haben hier angenommen, daß der obere (höherwertige) Teil des Operanden „über" dem unteren (niederwertigen) Teil gespeichert ist, d. h. bei der niedrigeren Speicheradresse. Dies braucht nicht notwendigerweise der Fall zu sein. Tatsächlich speichert der Z80 Adressen umgekehrt ab: Der untere Teil wird zuerst im Speicher abgelegt und der obere Teil dann in der nächsten Speicheradresse. Um sowohl für Daten als auch für Adressen eine gemeinsame Konvention zu verwenden, sei es empfohlen, auch bei den Daten den unteren Teil über dem oberen Teil abzulegen. Dies ist in Abb. 3.6 dargestellt.

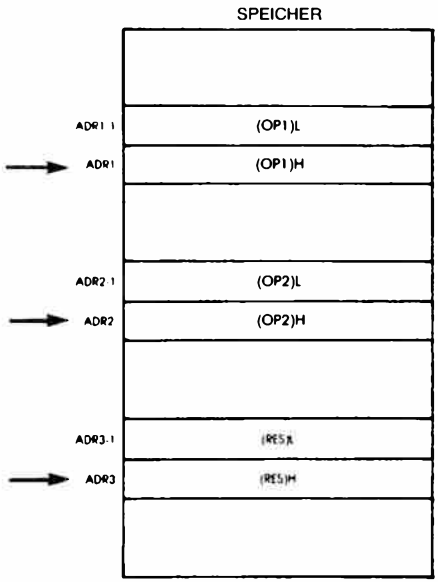


Abb. 3.7: Zeiger auf das obere Byte

Wenn man mit Operanden aus mehreren Bytes arbeitet, ist es wichtig, zwei wesentliche Vereinbarungen zu beachten:

- die Reihenfolge, in der die Daten im Speicher abgelegt werden,
- worauf die Datenzeiger zeigen, auf das obere oder das untere Byte.

Die Aufgaben 3.2 und 3.3 sollen diesen Punkt klären.

Aufgabe 3.2: Schreibe das obige Programm zur Addition von 16-Bit-Zahlen auf die Speicherbelegung um, die in Abb. 3.6 angegeben ist.

Aufgabe 3.3: Wir wollen annehmen, daß ADR1 nicht auf die untere Hälfte von OPR1 zeigt (wie in Abb. 3.5 und 3.6), sondern auf die obere Hälfte von OPR1. Dies ist in Abb. 3.7 veranschaulicht. Schreibe auch dazu das entsprechende Programm.

Der Programmierer muß entscheiden, wie 16-Bit-Zahlen gespeichert werden sollen (d. h. die untere oder die obere Hälfte zuerst) und ob die Zeiger auf die Adressen der unteren oder oberen Hälfte einer solchen Zahl zeigen sollen. Dies ist eine weitere Entscheidung, die Sie treffen müssen, wenn Sie Algorithmen oder Datenstrukturen entwerfen.

Die oben vorgestellten Programme sind herkömmliche Programme, die den Akkumulator benutzen. Wir werden jetzt ein anderes Programm zur 16-Bit-Addition vorstellen, das den Akkumulator nicht benutzt, statt dessen aber einige der speziellen 16-Bit-Befehle verwendet, die auf

dem Z80 verfügbar sind. Es sei angenommen, daß die Operanden so abgespeichert sind, wie in Abb.3.6 gezeigt. Das Programm lautet:

```
LD   HL,(ADR1) Lade HL mit OP1
LD   BC,(ADR2) Lade BC mit OP2
ADD  HL,BC     Addiere 16 Bit
LD   (ADR3),HL Speichere RES nach ADR3
```

Beachten Sie, wie viel kürzer dieses Programm im Vergleich mit unserem vorhergehenden Programm ist. Es ist „eleganter". *Auf eingeschränkte Art und Weise kann man die Register H und L beim Z80 als 16-Bit-Akkumulator verwenden.*

Aufgabe 3.4: Schreiben Sie unter Verwendung der 16-Bit-Befehle, die wir gerade eingeführt haben, ein Programm zur Addition von 32-Bit-Operanden. Die Operanden seien gespeichert wie in Abb. 3.8 dargestellt.

Lösung:

```
LD HL, (ADR1-1)
LD BC, (ADR2-1)
ADD HL,BC
LD (ADR3-1), HL
LD HL,(ADR1-3)
LD BC,(ADR2-3)
ADC HL,BC
LD (ADR3-3),HL
```

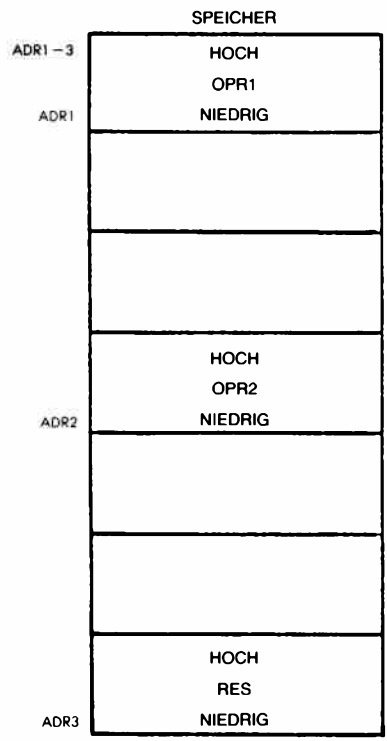


Abb. 3.8: Eine 32-Bit-Addition

Nachdem wir gelernt haben, wie man eine binäre Addition ausführt, wollen wir uns der Subtraktion zuwenden.

Subtraktion von 16-Bit-Zahlen

Eine 8-Bit-Subtraktion wäre zu einfach. Wir wollen sie als Übungsaufgabe aufheben und direkt eine 16-Bit-Subtraktion ausführen. Wie üblich sind unsere beiden Operanden OP1 und OP2 bei den Adressen ADR1 und ADR2 abgelegt. Es wird angenommen, daß die Speicherbelegung der nach Abb. 3.6 entspricht. Um zu subtrahieren, verwenden wir einen Subtraktionsbefehl (SBC) statt eines Additionsbefehls (ADD).

Aufgabe 3.5: Jetzt schreiben wir ein Subtraktionsprogramm.

Das Programm ist unten angegeben. Abb. 3.9 zeigt die Datenwege.

```
LD   HL,(ADR1)  OP1 nach HL
LD   DE,(ADR2)  OP2 nach DE
AND  A          Lösche Übertragsbit
SBC  HL,DE      OP1 – OP2
LD   (ADR3),HL  RES nach ADR3
```

Das Programm entspricht im wesentlichen dem, das wir zur 16-Bit-Addition entwickelt haben. Es gibt zwar zwei verschiedene Additionsbefehle für Registerpaare im Z80-Befehlssatz, ADD und ADC, aber nur einen Subtraktionsbefehl: SBC.

Deshalb muß man zwei Unterschiede beachten.

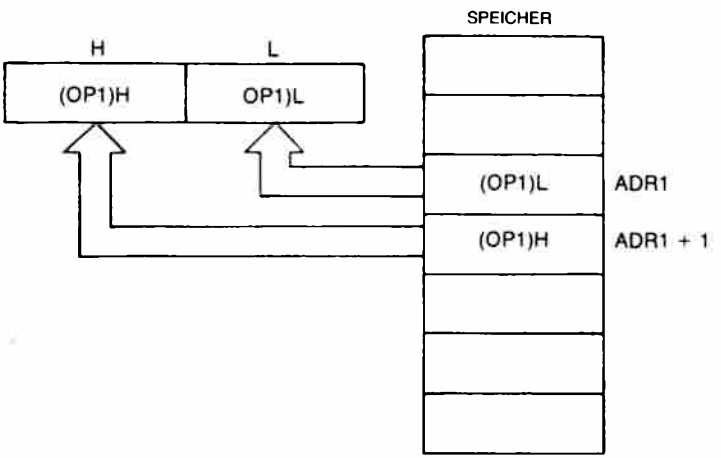


Abb. 3.9: 16-Bit-Ladebefehl – LD HL,(ADR1)

Die erste Veränderung ist die Verwendung von SBC statt ADD.

Der andere Unterschied ist der Befehl „AND A", der dazu dient, vor der Subtraktion das Übertragsbit zu löschen. Dieser Befehl verändert den Inhalt von A nicht.

Diese Vorsichtsmaßnahme ist nötig, da der Z80 auf zwei Arten in das Registerpaar HL addieren kann, nämlich mit und ohne Übertrag, aber nur auf eine Art vom Registerpaar HL subtrahieren kann, nämlich mit dem Befehl SBC „subtrahiere mit Übertrag". Da SBC automatisch auch das Übertragsbit subtrahiert, muß es vor der Subtraktion auf „0" gesetzt werden. Dies ist die Aufgabe des Befehls „AND A".

Aufgabe 3.6: Schreibe das Subtraktionsprogramm so um, daß keine speziellen 16-Bit-Befehle verwendet werden.

Aufgabe 3.7: Schreibe ein Subtraktionsprogramm für 8-Bit-Operanden.

Es sei daran erinnert, daß der endgültige Wert des Übertragsbits bei der Arithmetik von Zweierkomplementen ohne Bedeutung ist. Wenn bei der Subtraktion ein Überlauf auftritt, ist das Überlaufbit (Bit V) des Flagregisters gesetzt. Es kann dann getestet werden.

Die vorgestellten Beispiele sind einfache duale Additionen und Subtraktionen. Man kann jedoch auch einen anderen Typ von Arithmetik verwenden: die BCD-Arithmetik.

BCD-Arithmetik

8-Bit BCD Addition

Die Grundlagen der BCD-Arithmetik wurden in Kapitel 1 erläutert. Wir wollen uns ihre Eigenschaften nochmals ins Gedächtnis rufen. Sie wird im wesentlichen für kommerzielle Anwendungen verwendet, wo man alle signifikanten Stellen des Ergebnisses berücksichtigen muß. In der BCD-Darstellung wird ein 4-Bit-Nibble verwendet, um eine Dezimalziffer (0 bis 9) zu speichern. Deshalb kann jedes 8-Bit-Byte zwei BCD-Ziffern speichern. (Dies heißt *gepackte BCD-Darstellung*.) Wir wollen jetzt zwei Bytes addieren, von denen jedes zwei BCD-Ziffern enthält.

Um das Problem kennenzulernen, wollen wir zuerst einige Rechenbeispiele ausprobieren.

Wir wollen „01" und „02" addieren.

„01" wird dargestellt als: 0000 0001
„02" wird dargestellt als: 0000 0010

Das Ergebnis ist: 0000 0011

Dies ist die BCD-Darstellung von „03". (Wenn Sie in Bezug auf die BCD-Darstellung unsicher sind, verwenden Sie die Umwandlungstabelle am Ende des Buches.) In diesem Fall funktionierte alles sehr einfach. Wir wollen jetzt ein anderes Beispiel probieren.

„08" wird dargestellt als 0000 1000
„03" wird dargestellt als 0000 0011

Aufgabe 3.8: Berechne die Summe der beiden obigen Zahlen in BCD-Darstellung. Was kommt heraus? (Die Antwort folgt.)

Wenn Sie „0000 1011" erhielten, haben Sie die *duale* Summe von 8 und 3 berechnet. Dann haben Sie in der Tat *dual* 11 erhalten. Unglücklicherweise ist „1011" ein *unzulässiger Kode* in BCD. Es hätte sich die *BCD*-Darstellung von „11", d. h. „0001 0001" ergeben sollen!

Das erste Problem resultiert aus der Tatsache, daß die BCD-Darstellung nur die ersten zehn Kombinationen von vier Ziffern verwendet, um die Dezimalsymbole 0 bis 9 zu kodieren. Die restlichen sechs Kombinationen werden nicht verwendet, und die unzulässige „1011" ist eine dieser Kombinationen. Mit anderen Worten, immer wenn die Summe zweier BCD-Ziffern größer als 9 ist, muß man zum Ergebnis 6 addieren, um die 6 unbelegten Kodes zu überspringen.

Addiere die BCD-Darstellung von „6" zu 1011:

1011 (unzulässiger BCD-Kode)
+ 0110 (+6)

Das Ergebnis ist: 0001 0001

Dies ist in der Tat „11" in der BCD-Darstellung! Jetzt haben wir das richtige Ergebnis.

Dieses Beispiel zeigt eine der grundlegenden Schwierigkeiten des BCD-Modus. Man muß die sechs fehlenden Kodes ausgleichen. Man muß einen speziellen Befehl „DAA", genannt „dezimaler Abgleich", verwenden, um das Ergebnis der dualen Addition anzupassen. (Addiere 6, wenn das Ergebnis größer als 9 ist.)

Das nächste Problem wird am gleichen Beispiel dargestellt. In unserem Beispiel tritt ein Übertrag von der unteren BCD-Ziffer (der rechten) in die linke auf. Dieser interne Übertrag muß berücksichtigt und zur zweiten BCD-Ziffer addiert werden. Der Additionsbefehl macht dies automatisch. Es ist jedoch oft nützlich diesen internen Übertrag von Bit 3 nach Bit 4 (den „Halbübertrag") zu erkennen. Zu diesem Zweck gibt es das H-Flag.

Als Beispiel folgt hier ein Programm zur Addition der BCD-Zahlen „11" und „22":

```
LD   A,11H      Lade BCD-Zahl „11"
ADD  A,22H      Addiere BCD-Zahl „22"
DAA             Dezimalabgleich
LD   (ADR),A    Speichere Ergebnis
```

In diesem Programm verwenden wir das neue Symbol „H". Im Operandenfeld eines Befehls bedeutet das Zeichen „H", daß die davor stehenden Daten in hexadezimaler Form angegeben sind. Für die Ziffern „0" bis „9" sind hexadezimale und BCD-Darstellung identisch. Hier wollen wir die Literals (oder Konstanten) „11" und „22" addieren. Das Ergebnis wird bei der Adresse ADR abgelegt. Ist der Operand Teil des Befehls, wie im obigen Beispiel, dann nennt man dies *unmittelbare Adressierung* (die verschiedenen Adressierungsarten werden im Kapitel 5 genau diskutiert.) Das Speichern des Ergebnisses an einer festgelegten Adresse, wie bei LD (ADR),A, nennt man *absolute Adressierung*, wobei ADR eine 16-Bit-Adresse darstellt.

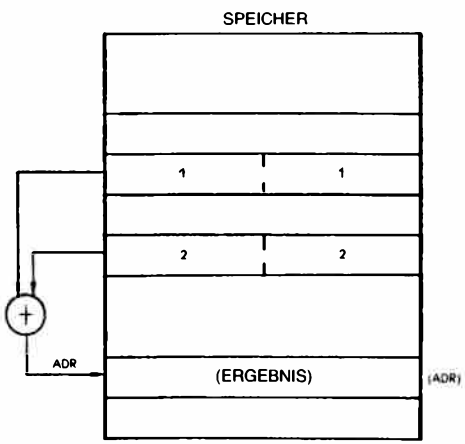


Abb. 3.10: Speicherung von BCD-Ziffern

Dieses Programm ähnelt dem für die duale 8-Bit-Addition, verwendet aber einen neuen Befehl: „DAA". Wir wollen seine Aufgabe an einem Beispiel veranschaulichen. Wir wollen zuerst „11" und „22" in BCD addieren:

```
  00010001   (11)
+ 00100010   (22)
-------------------
= 00110011   (33)
    3    3
```

Mit den Regeln der dualen Addition ergibt sich das richtige Ergebnis.

Wir wollen jetzt „22" und „39" addieren, indem wir die Regeln der *dualen* Addition verwenden:

```
  00100010   (22)
+ 00111001   (39)
-------------------
= 01011011
    5    ?
```

„1011" ist *kein zulässiger* BCD-Kode. Dies ist der Fall, weil BCD nur die ersten zehn Binärkodes benutzt und die nächsten sechs „überspringt". Wir müssen das gleiche tun, d. h. zum Ergebnis 6 addieren:

```
  01011011   (duales Ergebnis)
+ 00000110   (6)
-------------------
= 01100001   (61)
    6    1
```

Dies ist das richtige Ergebnis in BCD.

Aufgabe 3.9: Könnten wir den Befehl DAA in dem Programm hinter dem Befehl LD (ADR),A ausführen?

BCD Subtraktion

Die Subtraktion in BCD sieht kompliziert aus. Um eine Substraktion in BCD durchzuführen, muß man das *Zehnerkomplement* der Zahl addieren, genau so, wie man das Zweierkomplement addierte, um dual zu subtrahieren. Das Zehnerkomplement erhält man, indem man das Komplement zu 9 bildet und dann „1" addiert. Auf einem Standardmikroprozessor nimmt das normalerweise drei bis vier Operationen in Anspruch. Der Z80 jedoch besitzt einen leistungsfähigen DAA-Befehl, der das Programm vereinfacht. Der Befehl DAA korrigiert den Wert des Ergebnisses im Akkumulator automatisch in Abhängigkeit von den Flags C und H vor Ausführung von DAA. (Weitere Einzelheiten von DAA folgen im nächsten Kapitel.)

16-Bit-BCD-Addition

Die Addition von 16-Bit-Zahlen wird genauso einfach durchgeführt wie bei Dualzahlen. Das Programm für eine solche Addition ist unten angegeben:

```
LD   A,(ADR1)     Lade (OP1)L nach A
LD   HL,ADR2      Lade ADR2 nach HL
ADD  A,(HL)       (OP1 + OP2)L
DAA               Dezimalanpassung
LD   (ADR3),A     Speichere unteres Ergebnis
LD   A,(ADR+1)    Lade (OP1)H nach A
INC  HL           Zeiger auf ADR2+1
ADC  A,(HL)       (OP1 + OP2)H + Übertrag
DAA               Dezimalanpassung
LD   (ADR3+1),A   Speichere oberes Ergebnis
```

Gepackte BCD-Subtraktion

Die einfache Addition und Subtraktion von BCD-Zahlen haben wir beschrieben. Tatsächlich enthalten BCD-Zahlen aber eine beliebige Anzahl von Bytes. Als einfaches Beispiel für eine gepackte BCD-Subtraktion wollen wir annehmen, daß die beiden Zahlen, die in N1 und N2 zu finden sind, die gleiche Anzahl von BCD-Bytes enthalten. Die Zahl der Bytes wird COUNT genannt. Die Register und die Speicherbelegung ist in Abb. 3.11 dargestellt. Das Programm erscheint unten:

```
BCDPAK  LD    B,COUNT
        LD    DE,N2
        LD    HL,N1
        AND   A          Lösche Übertrag
MINUS   LD    A,(DE)     Byte N2
        SBC   A,(HL)
        DAA
        LD    (HL),A     Speichere Ergebnis
        INC   DE
        INC   HL
        DJNZ  MINUS      Dekrementiere B, Schleife bis B=0
```

N1 und N2 sind die Adressen, bei denen die BCD-Zahlen gespeichert sind. Diese Adressen werden in die Registerpaare DE und HL geladen:

```
BCDPAK  LD    B,COUNT
        LD    DE,N2
        LD    HL,N1
```

Im Gegensatz zur ersten Subtraktion muß dann das Übertragsbit gelöscht werden. Es wurde betont, daß das Übertragsbit auf verschiedene äquivalente Arten gelöscht werden kann. Hier benutzen wir beispielsweise:

```
        AND   A
```

Das erste Byte von N2 wird in den Akkumulator geladen, dann das erste

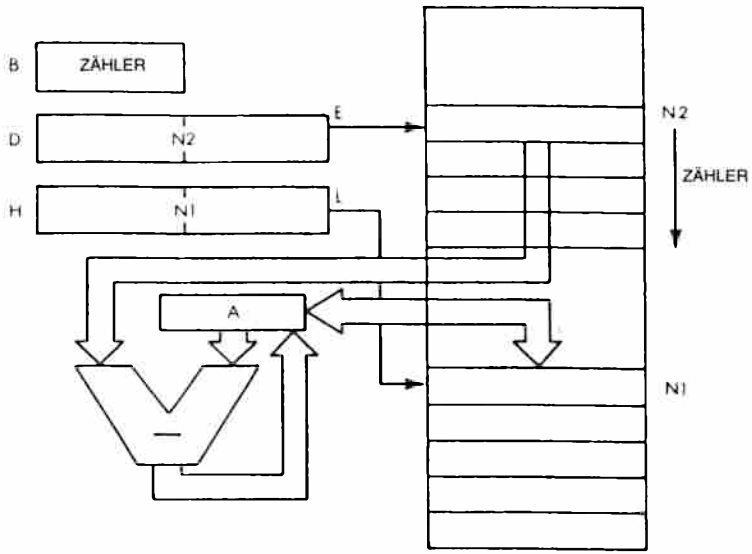


Abb. 3.11: Gepackte BCD-Subtraktion: N1←N2 − N1

Byte von N1 davon subtrahiert. Der Befehl DAA dient dann dazu, das richtige Ergebnis in BCD-Darstellung zu erhalten:

```
MINUS   LD    A,(DE)
        SBC   A,(HL)
        DAA
```

Das Ergebnis wird dann in N1 abgelegt:

```
        LD    (HL),A
```

Schließlich werden die Zeiger auf das aktuelle Byte inkrementiert:

```
        INC   DE
        INC   HL
```

Dann wird der Zähler dekrementiert und die Subtraktionsschleife so lange wiederholt, bis er den Wert „0" erreicht:

```
        DJNZ  MINUS
```

Der Befehl DJNZ ist ein spezieller Z80-Befehl, der in einem einzigen Befehl das Register B dekrementiert und den Sprung ausführt, wenn das Register B nicht Null ist.

Aufgabe 3.10: Vergleiche das obige Programm mit dem für die duale 16-Bit-Addition. Wo liegt der Unterschied?

Aufgabe 3.11: Kann man die Rolle von DE und HL vertauschen? (Hinweis: Beachte SBC.)

Aufgabe 3.12: Schreibe ein Subtraktionsprogramm für 16-Bit-BCD-Zahlen.

BCD Flags

Bei der BCD-Arithmetik zeigt das Übertragsbit bei einer Addition an, daß das Ergebnis größer als 99 ist. Dies ist anders als bei Zweierkomplementen, weil BCD-Ziffern jeweils dual dargestellt sind. Umgekehrt kennzeichnet das Übertragsflag bei der Subtraktion ein „Borgen".

Befehlstypen

Wir haben jetzt zwei Typen von Mikroprozessorbefehlen verwendet. Wir haben den Befehl LD verwendet, der den Akkumulator von einer Speicherstelle lädt oder seinen Inhalt bei der festgelegten Adresse ablegt. Dies ist ein Befehl zur *Datenübertragung.*

Dann haben wir *arithmetische Befehle* wie ADD, SUB, ADC und SBC verwendet. Sie bewirken Additions- und Subtraktionsoperationen. Weitere ALU-Befehle werden bald in diesem Kapitel vorgestellt.

In dem Mikroprozessor Z80 gibt es weitere Typen von Befehlen, die wir bisher noch nicht verwendet haben. Teilweise sind dies „Sprungbefehle", die die Reihenfolge verändern, in der ein Programm abgearbeitet wird. Dieser neue Befehlstyp wird im nächsten Beispiel eingeführt. Beachten Sie, daß man Sprungbefehle oft „Verzweigungen" nennt, wenn sie an eine Bedingung geknüpft sind, d. h. wenn im Programm eine logische Auswahl besteht. Das Wort „Verzweigung" stammt von der Analogie zu einem Baum und bedeutet eine Gabelung in der Darstellung des Programms.

Multiplikation

Wir wollen jetzt ein komplizierteres arithmetisches Problem betrachten: die Multiplikation von Dualzahlen. Um den Algorithmus für eine duale Multiplikation zu entwickeln, wollen wir damit beginnen, eine gewöhnliche dezimale Multiplikation zu untersuchen. Wir wollen 12 und 23 multiplizieren.

```
   12   (Multiplikand)
× 23    (Multiplikator)
-------------------------
   36   (Teilprodukt)
+ 24
-------------------------
= 276   (Endergebnis)
```

Die Multiplikation wird ausgeführt, indem man die rechte Ziffer des Multiplikators mit dem Multiplikanden multipliziert, d. h. „3 × 12". Das Teilprodukt ergibt „36". Dann multipliziert man die nächste Ziffer des Multiplikators, d. h. „2" mit „12". Das Ergebnis „24" wird dann zu dem Teilprodukt addiert.

Es wird aber noch eine Operation durchgeführt: 24 ist um eine Stelle *nach links versetzt.* Wir sagen, daß 24 um eine Stelle nach links verschoben wird. Genauso hätten wir sagen können, das Teilprodukt (36) sei vor der Addition um eine Stelle *nach rechts verschoben* worden.

Dann werden die beiden richtig verschobenen Zahlen addiert und die Summe ergibt sich zu 276. Dies ist einfach. Die duale Multiplikation wird auf die gleiche Art ausgeführt.

Wir wollen uns ein Beispiel anschauen. Wir wollen 3 x 5 multiplizieren:

```
(5)          101   (MPD = Multiplikand)
(3)      ×   011   (MPR = Multiplikator)
-------------------------------------------
             101   (PP = Partialprodukt)
            101
           000
-------------------------------------------
(15)       01111   (RES = Resultat)
```

Um die Multiplikaton richtig durchzuführen, gehen wir genauso vor, wie wir es oben taten. Die formale Darstellung dieses Algorithmus erscheint in Abb. 3.12. Sie stellt ein Flußdiagramm für diesen Algorithmus dar, unser erstes Flußdiagramm. Wir wollen es näher untersuchen.

Dieses Flußdiagramm stellt den Algorithmus, den wir gerade vorgestellt haben, symbolisch dar. Jedes Rechteck stellt eine Anweisung dar, die ausgeführt werden soll. Es wird in einen oder mehrere Programmbefeh-

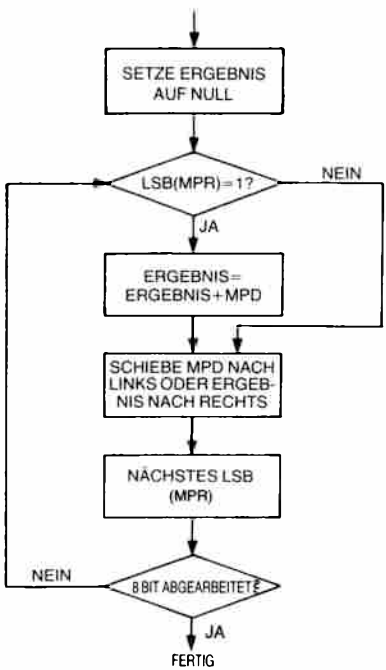


Abb. 3.12: Der grundlegende Algorithmus für die Multiplikation – ein Flußdiagramm

le übersetzt. Jede Raute stellt einen Test dar, der ausgeführt werden muß. Daraus wird im Programm eine *Verzweigung*. Fällt der Test positiv aus, führt die Verzweigung zu einer festgelegten Stelle. Hat der Test ein negatives Ergebnis, wird zu einer anderen Stelle verzweigt. Das Konzept der Verzweigung wird an späterer Stelle im Programm selbst erklärt. Der Leser sollte jetzt dieses Flußdiagramm selbst untersuchen und sich davon überzeugen, daß es den erläuterten Algorithmus wirklich genau beschreibt. Beachten Sie, daß aus der letzten Raute am unteren Ende des Flußdiagramms ein Pfeil herauskommt, der zur ersten Raute oben zurückführt. Der Grund dafür ist, daß der gleiche Teil des Flußdiagramms acht Mal ausgeführt wird, einmal für jedes Bit des Multiplikators. Einen solchen Vorgang, wo die Ausführung wieder an derselben Stelle beginnt, nennt man aus naheliegenden Gründen eine Programmschleife.

Aufgabe 3.13: Multiplizieren Sie „4" und „7" dual. Benutzen Sie dabei das Flußdiagramm und überprüfen Sie, ob Sie „28" erhalten. Versuchen Sie es nochmals, wenn dieses Ergebnis nicht herauskam. Nur wenn Sie das richtige Ergebnis erhalten, sind Sie in der Lage, das Flußdiagramm in ein Programm zu übersetzen.

8-mal-8-Bit Multiplikation

Wir wollen jetzt dieses Flußdiagramm in ein Programm für den Z80 umsetzen. Das vollständige Programm ist in Abbildung 3.13 wiedergegeben. Wir wollen es genau untersuchen. Wie Sie noch aus Kapitel 1 wissen werden, besteht Programmierung hier darin, das Flußdiagramm aus Abb. 3.12 in das Programm der Abb. 3.13 zu übersetzen. Jedes Kästchen des Flußdiagramms wird dabei in einen oder mehrere Befehle übersetzt.

Es wird angenommen, daß MPR und MPD schon eine Zahl enthalten.

```
MPY88   LD    BC,(MPRAD)   Lade Multiplikator nach C
        LD    B,8          B ist der Schleifenzähler
        LD    DE,(MPDAD)   Lade Multiplikand nach E
        LD    D,0          Lösche D
        LD    HL,0         Setze Ergebnis zu 0
MULT    SRL   C            Schiebe Multiplikatorbit ins
                           Übertragsbit
        JR    NC,NOADD     Teste Übertragsbit
        ADD   HL,DE        Addiere MPD zum Ergebnis
NOADD   SLA   E            Schiebe MPD nach links
        RL    D            Speichere Bit in D
        DEC   B            Dekrementiere Schleifenzähler
        JP    NZ,MULT      Wiederhole, bis Schleifenzähler = 0
        LD    (RESAD),HL   Speichere Ergebnis
```

Abb. 3.13: 8 x 8 Bit Multiplikationsprogramm

Das erste Kästchen unseres Flußdiagramms dient zur *Initialisierung*. Es ist nötig, bestimmte Register oder Speicherplätze auf „0" zu setzen, weil das Programm sie benutzen wird. Die Register, die das Programm verwendet, erscheinen in Abb. 3.14.

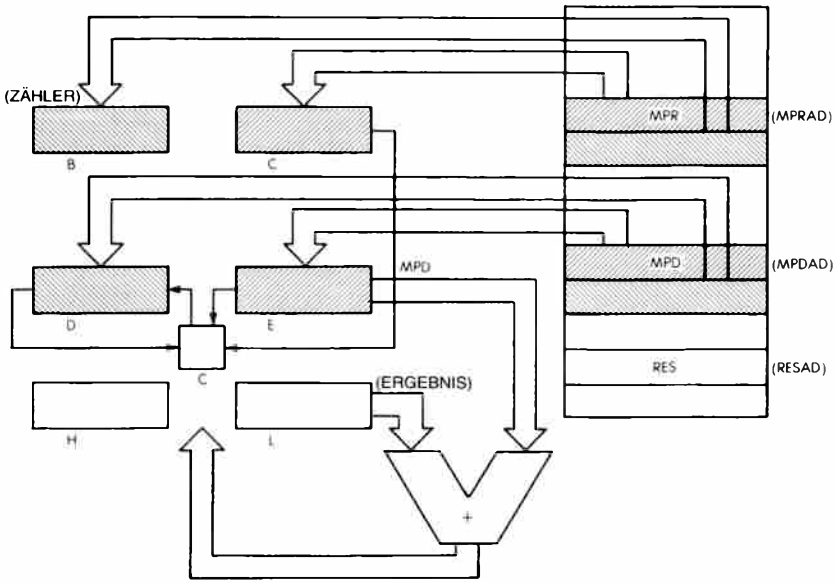


Abb. 3.14: 8 x 8 Bit Multiplikation – die Register

Für das Multiplikationsprogramm werden drei Registerpaare des Z80 verwendet. Der 8-Bit-Multiplikator stehe in der Speicherstelle MPRAD. Der Multiplikand sei bei der Speicheradresse MPDAD abgelegt. Multiplikator und Multiplikand werden jeweils in die Register C und E abgelegt (siehe Abb. 3.14). Das Register B wird als Zähler verwendet.

Die Register D und E enthalten den Multiplikanden, da er jeweils um ein Bit nach links geschoben wird.

Beachten Sie, daß man einen 16-Bit-Ladebefehl verwenden muß, obwohl anfangs nur C und E geladen werden müssen. Dadurch werden aber auch B und D aus dem Speicher geladen und müssen entsprechend auf „8" bzw. „0" zurückgesetzt werden.

Schließlich kann das Ergebnis einer 8-Bit x 8-Bit Multiplikation bis zu 16 Bit lang sein. Dies gilt, da $2^8 \times 2^8 = 2^{16}$. Für das Ergebnis müssen deshalb zwei Register reserviert werden. Dies sind die Register H und L, wie in Abb. 3.14 dargestellt.

Als erster Schritt müssen die Register B, C und E mit dem entsprechenden Inhalt geladen und das Ergebnis (das Teilprodukt) auf den Wert „0" initialisiert werden, wie es in dem Flußdiagramm in Abb. 3.12 festgelegt wurde. Dies wird mit den folgenden Befehlen erledigt:

```
MPY88   LD   BC,(MPRAD)
        LD   B,8
        LD   DE,(MPDAD)
        LD   D,0
        LD   HL,0
```

Die ersten drei Befehle laden MPR ins Registerpaar BE, den Wert „8" ins Register B und MPD ins Registerpaar DE. Da MPR und MPD 8-Bit-Worte sind, werden sie tatsächlich in die Register C bzw. E geladen, während die nächsten Worte im Speicher hinter MPR und MPD in die Register B und D geladen werden. Dies ist in Abb. 3.15 und 3.16 dargestellt. Der nächste Befehl setzt den Inhalt von D zu Null.

In diesem Multiplikationsprogramm wird der Multiplikand nach links verschoben, bevor er zum Ergebnis addiert wird (beachten Sie, daß es stattdessen wahlweise auch möglich wäre, das Ergebnis nach rechts zu verschieben, wie es im vierten Kästchen des Flußdiagramms Abb. 3.12 angegeben ist). Der Multiplikand MPD wird bei jedem Schritt in das Register D hineingeschoben. Das Register muß deshalb anfangs auf den Wert „0" gesetzt werden. Dies wird mit dem vierten Befehl erledigt. Schließlich setzt der fünfte Befehl die Register H und L in einem einzigen Befehl auf Null.

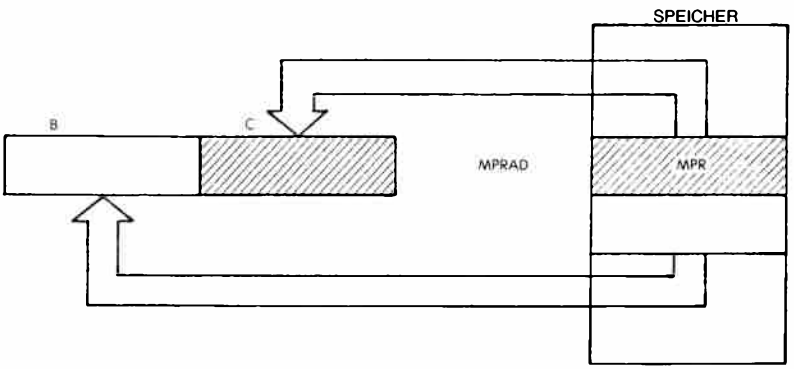


Abb. 3.15: LD BC,(MPRAD)

Wenn wir wieder das Flußdiagramm Abb. 3.12 betrachten, dann ist es der nächste Schritt, das niedrigste Bit (das Bit ganz rechts) des Multiplikators MPR zu testen. Ist dieses Bit eine „1", dann muß der Wert von MPD zum Zwischenergebnis addiert werden, sonst wird er nicht addiert. Dies wird mit den nächsten drei Befehlen durchgeführt:

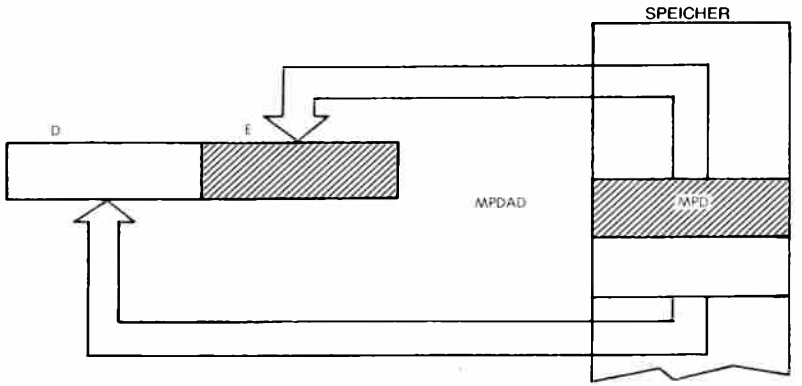


Abb. 3.16: LD DE,(MPDAD)

```
MULT    SRL  C
        JR   NC,NOADD
        ADD HL,DE
```

Das erste Problem, das wir lösen müssen, besteht darin, das niedrigste Bit des Multiplikators im Register C zu testen. Wir können hier den Befehl BIT des Z80 verwenden, mit dem man ein beliebiges Bit in einem beliebigen Register testen kann. In diesem Fall jedoch wollten wir ein möglichst einfaches Programm entwerfen, indem wir eine Schleife verwenden. Würden wir hier den Befehl BIT verwenden, dann müßten wir zuerst Bit 0 testen, danach Bit 1 usw., bis wir Bit 7 erreichten. Dazu wäre aber jedesmal ein anderer Befehl nötig und man könnte keine einfache Schleife verwenden. Um das Programm möglichst kurz zu halten, müssen wir einen anderen Befehl haben. Hier benutzen wir einen *Schiebebefehl.*

Anmerkung: Es gibt eine Möglichkeit, den Befehl BIT in einer Schleife zu verwenden, aber dann müßte das Programm sich selbst verändern, eine Methode, die wir vermeiden wollen..

SRL ist ein neuer Typ von Operationen in der Arithmetik- und Logikeinheit. Es bedeutet „Schiebe Rechts Logisch". Ein *logisches Schieben* wird dadurch charakterisiert, daß in die Position 7 eine Null nachgezogen wird. Im Gegensatz dazu ist das Bit, das beim *arithmetischen Schieben* in Position 7 gelangt, mit dem alten Wert von Bit 7 identisch. Die verschiedenen Arten von Schiebebefehlen werden im nächsten Kapitel beschrieben. Die Wirkungsweise des Befehls SRL C ist in Abb. 3.14 dargestellt, wo ein Pfeil aus dem Register C herauskommt und zu dem Quadrat führt, das das Übertragsbit (auch „C" genannt) darstellt. An dieser Stelle steht das rechte Bit von MPR im Übertragsbit C, wo es getestet werden kann.

Der nächste Befehl „JR NC,NOADD" ist ein Sprung. Er bedeutet „Springe, falls kein Übertrag (NC), zu der Adresse (Marke) NOADD". Ist der Inhalt des Übertragsbits eine „0" (kein Übertrag), dann springt das Programm zur Adresse NOADD. Enthält das Übertragsbit eine „1" (das Übertragsbit ist gesetzt), tritt keine Verzweigung ein und der nachfolgende Befehl, d. h. der Befehl „ADD HL,DE", wird ausgeführt.

Dieser Befehl legt fest, daß der Inhalt von D und E zum Inhalt von H und L addiert werden soll, mit dem Ergebnis in H und L. Da E den Multiplikanden MPD enthält (siehe Abb. 3.14), wird damit der Multiplikand zum Zwischenergebnis addiert.

An dieser Stelle muß der Multiplikand MPD nach links verschoben werden, unabhängig davon, ob er zum Zwischenergebnis addiert wurde oder nicht (dies entspricht dem vierten Kästchen im Fußdiagramm Abb. 3.12). Ausgeführt wird dies durch:

```
NOADD   SLA E
```

SLA bedeutet „Schiebe Links Arithmetisch". Es ist gerade oben erklärt worden, daß es zwei verschiedene Schiebebefehle gibt, ein logisches Schieben und ein arithmetisches Schieben. Dies ist das arithmetische Schieben. Beim Linksschieben legt SLA fest, daß das Bit, das rechts in das Register hereinkommt (das niedrigste Bit) eine „0" ist (wie vorher entsprechend beim SRL).

Wir wollen beispielsweise annehmen, daß das Register E anfangs den Inhalt 00001001 hat. Nach dem Befehl SLA E ist der Inhalt von E 00010010, und das Übertragsbit enthält eine „0".

Wenn wir jedoch wieder die Abbildung 3.14 betrachten, sehen wir, daß wir in Wirklichkeit das höchste Bit (MSB genannt = englisch: most significant bit) von E direkt nach D schieben wollen (dies ist durch einen Pfeil veranschaulicht, der von E zu D führt). Es gibt jedoch keinen Befehl, der ein Doppelregister wie D und E in einer Operation schiebt. Sobald der Inhalt von E verschoben wurde, ist das linke Bit ins Übertragsbit „herausgefallen". Wir müssen dieses Bit aus dem Übertragsbit holen und ins Register D hineinschieben. Dies wird mit dem nächsten Befehl ausgeführt:

```
        RL  D
```

RL ist noch eine andere Art von Schiebeoperation. Sie bedeutet „Rotiere Links". Bei einer *Rotieroperation* ist das Bit, das in das Register hereinkommt, der Inhalt des Übertragsbits, im Gegensatz zu einer *Schiebeoperation* (siehe Abb. 3.17). Dies ist genau das, was wir wollen. Der Inhalt des Übertragsbits wird ganz rechts ins Register D geladen, und wir haben tatsächlich das linke Bit des Registers E übertragen.

Diese Folge von zwei Befehlen ist in Abb. 3.18 dargestellt. Man sieht, daß das höchstwertige Bit von E, das durch ein X gekennzeichnet ist, zuerst ins Übertragsbit und dann an die niedrigste Stelle von D übertragen wird. Tatsächlich wurde es von E nach D verschoben.

Wenn wir uns wieder auf das Flußdiagramm Abb. 3.12 beziehen, müssen wir an dieser Stelle zum nächsten Bit von MPR gehen und testen, ob es das achte Bit ist. Dies geschieht, indem wir den Bytezähler im Register B dekrementieren (siehe Abb. 3.14). Das Register wird dekrementiert durch:

```
DEC  B
```

Dies ist ein *Dekrementierbefehl*, der die gewünschte Wirkung hat.

Schließlich müssen wir testen, ob der Zähler auf den Wert Null dekrementiert wurde. Dazu wird der Wert des Bits Z untersucht. Der Leser wird sich daran erinnern, daß das Flag Z (Nullflag) anzeigt, ob die vorangegangene arithmetische Operation (z. B. die Operation DEC) das Ergebnis Null ergeben hat. Man muß jedoch beachten, daß die Befehle DEC HL, DEC BC, DEC DE, DEC IX und DEC SP das Flag Z nicht beeinflussen. Ist der Zähler nicht „0", so ist die Operation noch nicht beendet, und wir müssen diese Programmschleife nochmals ausführen. Dies wird mit dem folgenden Befehl erledigt:

```
JP  NZ,MULT
```

Hier handelt es sich um einen Sprungbefehl, der festlegt, daß immer dann, wenn das Z-Bit nicht gesetzt ist (NZ bedeutet nicht Null), zur Adresse MULT gesprungen wird. Dies ist die *Programmschleife*, die wiederholt ausgeführt wird, bis B auf den Wert 0 dekrementiert wird. Sobald B auf den Wert 0 dekrementiert wurde, wird das Z-Bit gesetzt, und der Befehl JP NZ wird nicht ausgeführt. Dann wird aber der nächste folgende Befehl abgearbeitet, nämlich:

```
LD (RESAD),HL
```

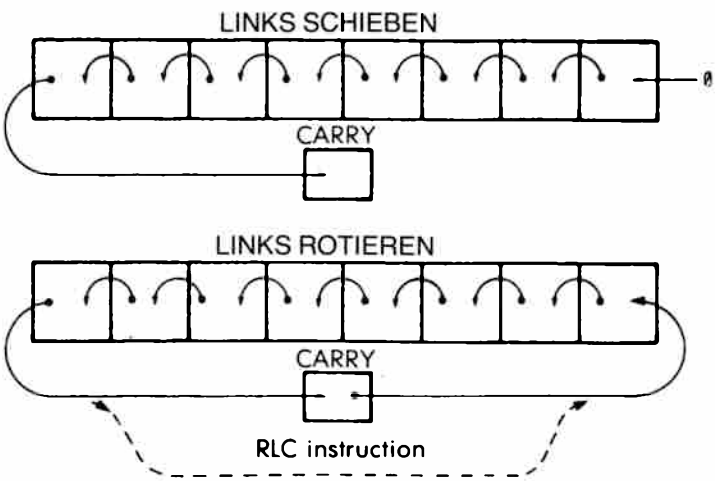


Abb. 3.17: Schieben und Rotieren

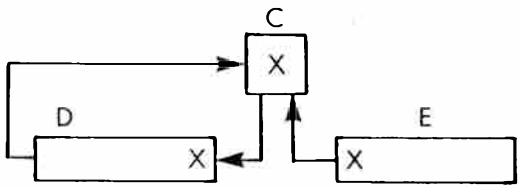


Abb. 3.18: von E nach D Schieben

Dieser Befehl speichert den Inhalt von H und L, d. h. das Ergebnis der Multiplikation, bei der Adresse RESAD, der Adresse, die für das Ergebnis festgelegt worden war. Beachten Sie, daß dieser Befehl den Inhalt der beiden Register H und L in zwei aufeinanderfolgende Speicherzellen transferiert, die den Adressen RESAD und RESAD + 1 entsprechen. Er speichert 16 Bit auf einmal ab.

Aufgabe 3.14: Können Sie das Multiplikationsprogramm so umschreiben, daß der Befehl BIT (Beschreibung im nächsten Kapitel) statt des Befehls SRL C verwendet wird? Was wäre der Nachteil davon?

Wir wollen das Programm wenn möglich verbessern:

Aufgabe 3.15: Kann der Befehl JP am Ende des Programms durch JR ersetzt werden? Wenn ja, welchen Vorteil würde dies bringen?

Aufgabe 3.16: Können Sie den Befehl DJNZ verwenden, um das Programm am Ende zu verkürzen?

Aufgabe 3.17: Überprüfen Sie die beiden Befehle „LD D,0" und „LD HL,0" am Anfang des Programms. Kann man sie ersetzen durch:

```
XOR A
LD D,A
LD H,A
LD L,A
```

Wenn ja, was ist der Gewinn an Länge (Anzahl der Bytes) und an Geschwindigkeit?

Es sei angemerkt, daß das Programm, das wir gerade entwickelt haben, meistens ein Unterprogramm sein wird, und daß der letzte Befehl in einem Unterprogramm RET (Return, d. h. Rücksprung) ist. Der Mechanismus von Unterprogrammen wird später in diesem Kapitel erklärt.

Wichtiger Selbsttest

Dies ist das erste wichtige Programm, mit dem wir uns bisher befaßt haben. Es enthält viele verschiedene Typen von Befehlen, so u. a. Transferbefehle (LD), arithmetische Operationen (ADD), logische Operationen (SRL, SLA, RL) und Sprünge (JP, JR). Es enthält außerdem eine Programmschleife, in der sieben Operationen, beginnend bei der

MARKE	BEFEHL	B	C	C (CARRY)	D	E	H	L

Abb. 3.19: Tabelle zur Multiplikationsaufgabe

Adresse MULT, wiederholt ausgeführt werden. Wenn man das Programmieren verstehen will, dann ist es wesentlich, die Arbeitsweise eines solchen Programms in allen Einzelheiten zu verstehen. Das Programm ist wesentlich länger als die vorhergehenden einfachen Arithmetikprogramme, die wir bisher entwickelt haben, und es sollte in allen Einzelheiten studiert werden. Jetzt wird eine wichtige Übungsaufgabe vorgeschlagen. Dem Leser sei es dringend empfohlen, diese Aufgabe vollständig und richtig durchzuführen, bevor er weitermacht. Dies alleine ist dann ein tatsächlicher Beweis, daß die Konzepte, die bisher vorgestellt wurden, auch verstanden sind. Wenn das richtige Ergebnis herauskommt, dann heißt das, daß Sie wirklich den Mechanismus verstanden haben, wie Befehle im Mikroprozessor Information verändern, zwischen Speicher und Registern transportieren und bearbeiten. Wenn Sie nicht das richtige Ergebnis erhalten, oder wenn Sie diese Aufgabe nicht

lösen, dann werden Sie wahrscheinlich später Schwierigkeiten haben, wenn Sie selbst Programme schreiben wollen. Um Programmieren zu lernen, muß man selbst üben. Bitte machen Sie jetzt eine Unterbrechung, nehmen Sie ein Blatt Papier oder verwenden Sie die Abbildung 3.19 und machen Sie die folgende Übungsaufgabe:

Aufgabe 3.18: Immer wenn man ein Programm geschrieben hat, sollte man es von Hand überprüfen,um sich davon zu überzeugen, daß es richtige Ergebnisse liefert. Genau das wollen wir jetzt tun: Das Ziel dieser Aufgabe ist es, die Tabelle in Abb. 3.19 vollständig und richtig auszufüllen.

Sie können direkt in die Abb. 3.19 schreiben, oder sie können eine Kopie des Formulars verwenden. Sie sollen den Inhalt aller wesentlichen Register im Z80 nach der Ausführung jedes Befehls in dem Programm angeben, vom Anfang bis zum Ende. In Abb. 3.19 sind alle Register angegeben. Von links nach rechts sind dies die Register B und C, die Register D, E, H und L. Auf der linken Seite der Tabelle soll die Marke, wenn vorhanden, und dann der Befehl, der ausgeführt wurde, angegeben werden. Rechts von dem Befehl soll der Inhalt aller Register nach der Ausführung des Befehls eingetragen werden. Wenn der Inhalt eines Registers nicht bekannt (oder nicht definiert) ist, können Sie das durch einen Strich kennzeichnen. Wir wollen gemeinsam anfangen, die Tabelle auszufüllen. Dann müssen Sie die Tabelle selbst bis zum Ende ausfüllen. Die erste Zeile ist unten angegeben:

MARKE	BEFEHL	B	C	C	D	E	H	L
		--	--	-	--	--	--	--
MPY88	LD BC,(0200)	--	03	-	--	--	--	--

Abb. 3.20: Multiplikation: Nach dem ersten Befehl

Wir wollen hier annehmen, daß wir „3" (MPR) und „5" (MPD) multiplizieren.

Der Befehl, der als erster ausgeführt werden muß, ist „LD BC,(MPRAD)". Der Inhalt der Speicherstelle MPRAD wird in die Register B und C geladen. Es wurde angenommen, daß MPR gleich 3 ist, d. h. gleich „00000011". Nach Ausführung dieses Befehls wurde der Inhalt des Registers auf „3" gesetzt. Beachten Sie, dieser Befehl bewirkt auch, daß das Register B mit dem Inhalt geladen, der im Speicher auf MPR folgt. Der nächste Befehl im Programm berücksichtigt das, indem er das Register B mit „8" lädt, wie in Abb. 3.21 gezeigt. Beachten Sie, daß an dieser Stelle der Inhalt der Register D, E, H und L noch undefi-

niert ist. Dies wird durch Striche markiert. Der Befehl LD legt das Übertragsbit nicht fest, so daß auch der Inhalt des Übertragsbits nicht definiert ist. Auch dies ist durch einen Strich kenntlich gemacht.

MARKE	BEFEHL	B	C	C	D	E	H	L
		--	--	-	--	--	--	--
MPY88	LD BC,(0200)	00	03	-	--	--	--	--
	LD B, 08	08	03	-	--	--	--	--

Abb. 3.21: Multiplikation: Nach zwei Befehlen

Die Situation nach Ausführung der ersten fünf Befehle (bis direkt vor MULT) ist in Abb. 3.22 gezeigt.

MARKE	BEFEHL	B	C	C	D	E	H	L
		--	--	-	--	--	--	--
MPY88	LD BC,(0200)	00	03	-	--	--	--	--
	LD B, 08	08	03	-	--	--	--	--
	LD DE,(0202)	08	03	-	00	05	--	--
	LD D, 00	08	03	-	00	05	--	--
	LD HL,0000	08	03	-	00	05	00	00

Abb. 3.22: Multiplikation: Nach fünf Befehlen

Der Befehl SRL führt ein logisches Rechtsschieben aus, und das rechte Bit von MPR gelangt in das Übertragsbit. In Abb. 3.23 können Sie sehen, daß MPR nach dem Schieben den Inhalt „0000 0001" hat. Das Übertragsbit C ist jetzt auf „1" gesetzt. Die anderen Register werden durch diese Operation nicht verändert. Bitte füllen Sie die Tabelle jetzt selbst weiter aus.

Ein zweiter Durchlauf durch die Schleife ist am Ende dieses Kapitels in Abb. 3.41 dargestellt.

Eine komplette Liste, die die Inhalte aller Z80-Register und Flags enthält, zeigt die Abbildung 3.39 am Ende des Kapitels für die vollständige Multiplikation. Abb. 3.40 zeigt eine Assemblerliste des Programms.

MARKE	BEFEHL	B	C	C	D	E	H	L
		--	--	-	--	--	--	--
MPY88	LD BC,(0200)	00	03	-	--	--	--	--
	LD B, 08	08	03	-	--	--	--	--
	LD DE,(0202)	08	03	-	00	05	--	--
	LD D, 00	08	03	-	00	05	--	--
	LD HL,0000	08	03	-	00	05	00	00
MULT	SRL C	08	01	1	00	05	00	00
	JR NC,0114	08	01	1	00	05	00	00
	ADD HL,DE	08	01	0	00	05	00	05
NOADD	SLA E	08	01	0	00	0A	00	05
	RL D	08	01	0	00	0A	00	05
	DEC B	07	01	0	00	0A	00	05
	JP NZ,010F	07	01	0	00	0A	00	05

Abb. 3.23: Ein Durchlauf durch die Schleife

Andere Möglichkeiten der Programmierung

Das Programm, das wir eben entwickelt haben, hätte auch auf viele andere Arten geschrieben werden können. Es ist eine allgemeingültige Regel, daß normalerweise jeder Programmierer Wege finden kann, wie er ein Programm ändern, oft sogar verbessern kann. Beispielsweise haben wir den Multiplikanden vor der Addition nach links verschoben. Mathematisch wäre es äquivalent gewesen, das Zwischenergebnis vor der Addition des Multiplikanden um eine Position nach rechts zu verschieben. Dies ist tatsächlich eine interessante Übungsaufgabe!

Aufgabe 3.19: Schreiben Sie ein Programm zur 8 x 8 Bit Multiplikation, das den gleichen Algorithmus verwendet, aber das Ergebnis um eine Stelle nach rechts verschiebt, und nicht den Multiplikator um eine Stelle nach links. Vergleichen Sie es mit dem vorhergehenden Programm und entscheiden Sie, ob diese andere Ausführung schneller oder langsamer als das alte Programm ist. Die Ausführungsgeschwindigkeiten der Z80-Befehle sind im nächsten Kapitel angegeben.

Verbessertes Multiplikationsprogramm

Das Programm, das wir gerade entwickelt haben, war eine direkte Übersetzung des Algorithmus in Kode. *Effektives Programmieren erfordert jedoch eine genaue Beachtung der Einzelheiten,* und oft kann man die

Länge eines Programms verkleinern oder die Ausführungszeit verkürzen. Wir wollen jetzt andere Möglichkeiten untersuchen, wie wir dieses grundlegende Programm verbessern können.

1. Schritt

Eine erste mögliche Verbesserung liegt in der besseren Ausnutzung des Z80-Befehlssatzes. Der zweitletzte Befehl und der vorhergehende können durch einen einzigen Befehl ersetzt werden:

```
        DJNZ  LOOP
```

Dies ist ein spezieller „automatischer Sprung" des Z80, der das Register B dekrementiert und zu einer festgelegten Adresse verzweigt, wenn es nicht „0" ist. Um ganz genau zu sein, der Befehl ist nicht völlig identisch mit den vorhergehenden beiden Befehlen:

```
        DEC   B
        JP    NZ,MULT,
```

da er eine Distanz angibt und man nur innerhalb eines Bereichs von −128 bis +127 springen kann. Hier müssen wir jedoch zu einer Stelle springen, die nur einige Byte entfernt ist, so daß diese Verbesserung zulässig ist. Das Programm, das sich daraus ergibt, ist unten in Abb. 3.24 aufgelistet:

```
MPY88B LD    DE,(MPDAD)
       LD    BC,(MPRAD)
       LD    B,8
       LD    D,O
       LD    HL,0
MULT   SRL   C
       JR    NC,NOADD
       ADD   HL,DE
NOADD  SLA   E
       RL    D
       DJNZ  MULT
       LD    (RESAD),HL
       RET
```

Abb. 3.24: Verbesserte Multiplikation, Schritt 1

2. Schritt

Um dieses Multiplikationsprogramm weiter zu verbessern, wollen wir besonders beachten, daß in dem ursprünglichen Programm nach Abb. 3.13 drei verschiedene Schiebebefehle verwendet werden. Zuerst wird der Multiplikator nach rechts geschoben und dann in zwei verschiedenen Operationen der Multiplikand MPD nach links, indem zuerst das Register E nach links geschoben und dann das Register D nach links rotiert wird. Dies kostet Zeit. Ein üblicher „Trick" bei der Programmierung ba-

siert auf folgender Beobachtung: Jedesmal, wenn der Multiplikator um eine Position verschoben wird, wird eine weitere Bitposition im Multiplikatorregister frei. Wird der Multiplikator beispielweise nach rechts verschoben (wie im vorhergehenden Beispiel), wird links eine Stelle frei. Gleichzeitig kann man sehen, daß das erste Teilprodukt (oder Zwischenergebnis) höchstens neun Bit belegt. Wurde dem Ergebnis bei Beginn des Programms nur ein Byte zugewiesen, könnten wir jetzt die Stelle, die der Multiplikator frei macht, verwenden, um das Bit zu speichern, das aus dem Ergebnisbyte herausfällt.

Wird der MPR das nächste Mal verschoben, ist die Länge des Zwischenergebnisses wieder um höchstens ein Bit gewachsen. Mit anderen Worten, es genügt, für das Ergebnis anfangs nur ein Byte vorzusehen, und dann die Stellen zu benutzen, die frei werden, wenn der Multiplikator verschoben wird. Um das Programm zu verbessern, werden wir deshalb für MPR und RES ein gemeinsames Registerpaar vorsehen. Idealerweise sollten die beiden Register in einer Operation zusammen geschoben werden. Leider kann der Z80 immer nur 8 Bit auf einmal verschieben. Wie die meisten anderen 8-Bit-Mikroprozessoren besitzt er keine Befehle, die es erlauben, 16 Bit gleichzeitig zu verschieben.

Man kann jedoch einen anderen Trick anwenden. Der Z80 (wie auch der 8080) besitzt spezielle 16-Bit-Additionsbefehle, die wir schon benutzt haben. Vorausgesetzt, daß Multiplikator und Ergebnis im Registerpaar HL gespeichert sind, können wir folgenden Befehl verwenden:

```
ADD HL,HL,
```

der den Inhalt von HL mit sich selbst addiert. Eine Zahl mit sich selbst addieren heißt sie verdoppeln. Eine Zahl im Dualsystem verdoppeln

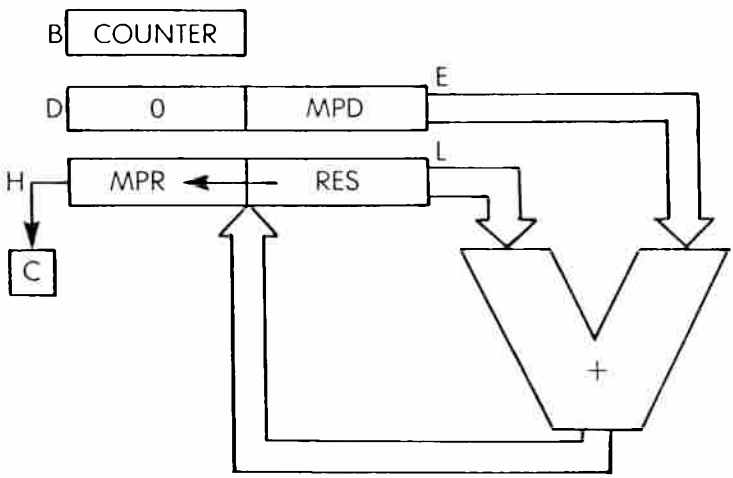


Abb. 3.25: Register zur verbesserten Multiplikation

heißt aber, sie nach links zu verschieben. Wir haben gerade ein 16-Bit-Schieben in einem einzigen Befehl gefunden. Leider wird nach links verschoben, während wir nach rechts schieben wollten. Dies ist aber kein Problem.

Prinzipiell kann der MPR nach links oder nach rechts verschoben werden. Wir haben einen Algorithmus verwendet, der nach rechts schiebt, weil man das bei der normalen Addition auch macht. Dies ist aber nicht zwingend. Die Additionsoperation ist kommutativ und die Reihenfolge kann vertauscht werden. Den MPR nach links zu verschieben ist also genauso richtig.

Um den Vorteil dieses simulierten 16-Bit-Schiebens zu nutzen, müssen wir den MPR nach links verschieben. Deshalb wird der MPR im Register H stehen und das Ergebnis im Register L. Daraus ergibt sich die Registerbelegung nach Abb. 3.25.

Der Rest des Programms ist im wesentlichen identisch mit der vorhergehenden Version. Es ergibt sich dann folgendes Programm:

```
MP488C LD    HL,(MPRAD-1)
       LD    L,0
       LD    DE,(MPDAD)
       LD    D,0
       LD    B,8
MULT   ADD   HL,HL
       JR    NC,NOADD
       ADD   HL,DE
NOADD DJNZ   MULT
       LD    (RESAD),HL
       RET
```

Abb. 3.26: Verbesserte Multiplikation, Schritt 2

Wenn wir dieses Programm mit dem vorhergehenden vergleichen, dann sehen wir, daß sich die Länge der Multiplikationsschleife (die Zahl der Befehle zwischen MULT und dem Sprung) verringert hat. Dieses Programm wurde mit weniger Befehlen geschrieben und wird deshalb auch schneller ausgeführt werden. Dies zeigt, wie vorteilhaft es ist, die richtigen Register zum Speichern der Information zu wählen.

Ein direkter Entwurf führt im allgemeinen zu einem Programm, das funktioniert. Er wird aber nicht zu einem *optimalen* Programm führen. Deshalb ist es wichtig, die verfügbaren Register und Befehle zu verstehen und auf die bestmögliche Art einzusetzen. Diese Beispiele veranschaulichen einen vernünftigen Weg zu einer Auswahl von Registern und Befehlen für maximale Effizienz.

Aufgabe 3.20: Berechnen Sie die Geschwindigkeit einer Multiplikation unter Verwendung des letzten Programms. Nehmen Sie an, daß in 50% der Fälle eine Verzweigung auftritt. Schauen Sie nach, wieviele Zyklen jeder Befehl beansprucht. Nehmen Sie eine Taktfrequenz von 2 MHZ an (d. h. ein Taktzyklus = 0,5 μs).

Aufgabe 3.21: Beachten Sie, daß wir für den Multiplikanden das Registerpaar DE verwendet haben. Wie müßte das Pogramm verändert werden, wenn wir stattdessen das Registerpaar BC verwendet hätten? (Hinweis: Dann wäre eine Veränderung am Ende nötig.)

Aufgabe 3.22: Warum müssen wir den Umstand auf uns nehmen, das Register D Null zu setzen, nachdem wir MPD nach E geladen haben?

Zuletzt wollen wir noch eine Einzelheit besprechen, über die sich der Programmierer, der mit dem Befehlssatz des Z80 noch nicht vertraut ist, wahrscheinlich wundert. Der Leser hat bestimmt bemerkt, daß wir beide Register D und E gleichzeitig von einer Speicheradresse laden mußten, um MPD aus dem Speicher nach E zu laden. Dies liegt daran, daß es keine Möglichkeit gibt, ein einzelnes Byte direkt aus dem Speicher zu holen und ins Register E zu laden, außer wenn die Adresse in HL steht. Dies ist eine Eigenart, die von dem frühen 8008 übernommen wurde, der keine direkte Adressierung zuließ. Diese Eigenschaft wurde mit einigen Verbesserungen in den 8080 übernommen und noch weiter verbessert

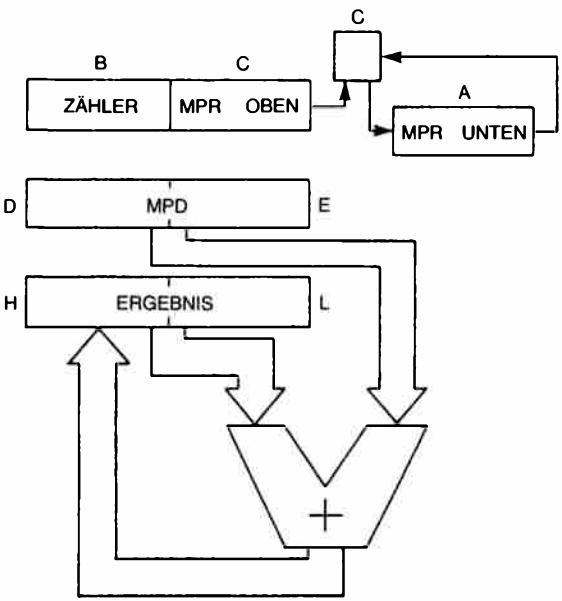


Abb. 3.27: 16 x 16 Bit Multiplikation – die Register

im Z80, bei dem man 16 Bit direkt von einer gegebenen Speicheradresse holen kann (jedoch keine 8 Bit).
Nachdem wir jetzt dieses mögliche Rätsel gelöst haben, wollen wir eine kompliziertere Multiplikation ausführen.

Eine 16 x 16 Bit Multiplikation

Um unsere neu erworbenen Fähigkeiten zu überprüfen, wollen wir zwei 16-Bit-Zahlen multiplizieren. Wir wollen jedoch annehmen, daß das Ergebnis nur 16 Bit einnimmt, so daß es in eines der Registerpaare paßt.
Wie bei unserem ersten Beispiel einer Multiplikation, steht das Ergebnis in den Registern H und L (siehe Abb. 3.27). Der Multiplikand MPD steht in den Registern D und E.

Es wäre verlockend, den Multiplikator in den Registern B und C abzulegen. Wenn wir jedoch den Vorteil des Befehls DJNZ nutzen wollen, muß das Register B als Zähler verwendet werden. Deshalb wird die eine Hälfte des Multiplikators im Register C stehen und die andere im Register A (siehe Abb. 3.27). Das Multiplikationsprogramm ist unten angegeben:

```
MUL16   LD    A,(MPRAD+1)   MPR, oben
        LD    C,A
        LD    A,(MPRAD)     MPR, unten
        LD    B,16D         Zähler
        LD    DE,(MPDAD)    MPD
        LD    HL,0
MULT    SRL   C             MPR oben nach
                            rechts schieben
        RRA                 MPR unten
                            rechts rotieren
        JR    NC,NOADD      Teste Übertrag
        ADD   HL,DE         Addiere MPD zum
                            Ergebnis
NOADD   EX    DE,HL         Vertausche DE, HL
        ADD   HL,HL         Verdopplung –
                            schiebe MPD links
        EX    DE,HL
        DJNZ  MULT
        RET
```

Abb. 3.28: 16 x 16 Bit Mutiplikationsprogramm

Dieses Programm ist analog zu dem, das wir vorher entwickelt haben. Die ersten sechs Befehle (von der Marke MUL16 bis zur Marke MULT) initialisieren die Register mit den entsprechenden Inhalten. Eine neue

Schwierigkeit taucht hier dadurch auf, daß die beiden Hälften von MPR mit verschiedenen Operationen geladen werden müssen. Es wurde angenommen, daß MPRAD auf den unteren Teil von MPR im Speicher zeigt, auf den in der nächsten Speicherstelle der obere Teil folgt. (Beachten Sie, daß auch die umgekehrte Konvention benutzt werden kann.) Sobald der obere Teil von MPR nach A eingelesen wurde, muß er nach C übertragen werden:

```
LD   A,(MPRAD+1)
LD   C,A
```

Schließlich kann der untere Teil von MPR direkt in den Akkumulator geladen werden:

```
LD   A,(MPRAD)
```

Die restlichen Register B, D, E, H und L werden wie üblich initialisiert:

```
LD   B,16D
LD   DE,(MPDAD)
LD   HL,0
```

Mit dem Multiplikator muß ein 16-Bit-Schieben ausgeführt werden. Es nimmt zwei getrennte Schiebe- oder Rotieroperationen mit den Registern C und A in Anspruch:

```
MULT  SRL  C
      RRA
```

Nach dem 16-Bit-Schieben steht das rechte Bit von MPR, d. h. das LSB, im Übertragsbit C, wo es getestet werden kann:

```
JR   NC,NOADD
```

Wie üblich wird der Multiplikand nicht zum Zwischenergebnis addiert, wenn das Übertragsbit „0" ist; der Multiplikand wird aber addiert, wenn es „1" ist:

```
ADD  HL,DE
```

Dann muß der Multiplikand MPD um eine Position nach links geschoben werden.

Der Z80 hat aber keinen Befehl, der den Inhalt der Register D und E gleichzeitig um eine Position nach links verschiebt, und ebensowenig kann er den Inhalt von D und E mit sich selbst addieren. Der Inhalt von D und E wird deshalb nach H und L übertragen, dann verdoppelt und nach D und E zurückübertragen. Dies wird mit den drei folgenden Befehlen durchgeführt:

```
NOADD  EX   DE,HL
       ADD  HL,HL
       EX   DE,HL
```

Schließlich wird der Zähler B dekrementiert und ein Sprung zum Schleifenanfang ausgeführt, wenn er nicht auf „0" dekrementiert wurde:

```
DJNZ  MULT
```

Wie üblich ist es möglich, die Register anders zu belegen, was eventuell (oder eventuell auch nicht) zu kürzerem Kode führen kann:

Aufgabe 3.23: Lade den Multiplikator in die Register B und C. Plaziere den Zähler in A. Schreibe das entsprechende Multiplikationsprogramm und diskutiere Vor- und Nachteile dieser Registerbelegung.

Aufgabe 3.24: Diese Aufgabe bezieht sich auf das ursprüngliche 16-Bit-Multiplikationsprogramm. Können Sie eine Möglichkeit angeben, den MPD zu schieben, der in den Registern D und E steht, ohne ihn in die Register H und L zu übertragen?

Aufgabe 3.25: Schreibe ein 16 mal 16 Bit Multiplikationsprogramm, das erkennt, wenn das Ergebnis länger als 16 Bit ist. Dies ist eine einfache Verbesserung unseres grundlegenden Programms.

Aufgabe 3.26: Schreibe ein 16 mal 16 Bit Multiplikationsprogramm mit einem 32-Bit-Ergebnis. In Abb. 3.29 ist dazu eine Registerbelegung vorgeschlagen. Beachten Sie, daß das Zwischenergebnis nach der ersten Addition in der Schleife nur 16 Bit einnimmt, und daß der Multiplikator für jeden weiteren Schritt ein Bit freimacht.

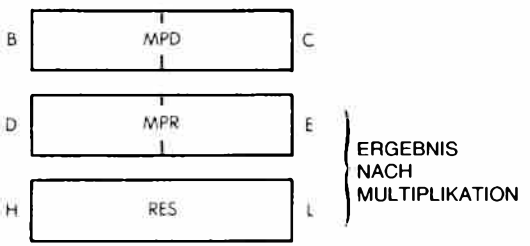


Abb. 3.29: 16 x 16 Bit Multiplikation mit 32-Bit-Ergebnis

Duale Division

Der Algorithmus für die duale Division ist analog zu dem, den wir für die Multiplikation verwendet haben. Der Divisor wird nacheinander von den Bits hoher Ordnung des Dividenden subtrahiert. Nach jeder Subtraktion wird statt des ursprünglichen Dividenden das Ergebnis benutzt. Der Wert des Quotienten wird gleichzeitig jedesmal um 1 inkrementiert. Eventuell wird das Ergebnis der Subtraktion negativ. Dies nennt man eine *Überziehung*. Dann muß man das Zwischenergebnis wiederherstellen, indem man den Divisor wieder addiert. Natürlich muß gleichzeitig der Quotient um 1 dekrementiert werden. Quotient und Dividend werden dann um eine Position nach links geschoben und der Algorithmus wird wiederholt. Abb. 3.30 zeigt das Flußdiagramm dazu.

Das eben beschriebene Verfahren nennt man *wiederherstellendes Verfahren*. Eine Variation dieses Verfahrens, die eine schnellere Ausführung bewirkt, heißt *nicht-wiederherstellendes Verfahren*.

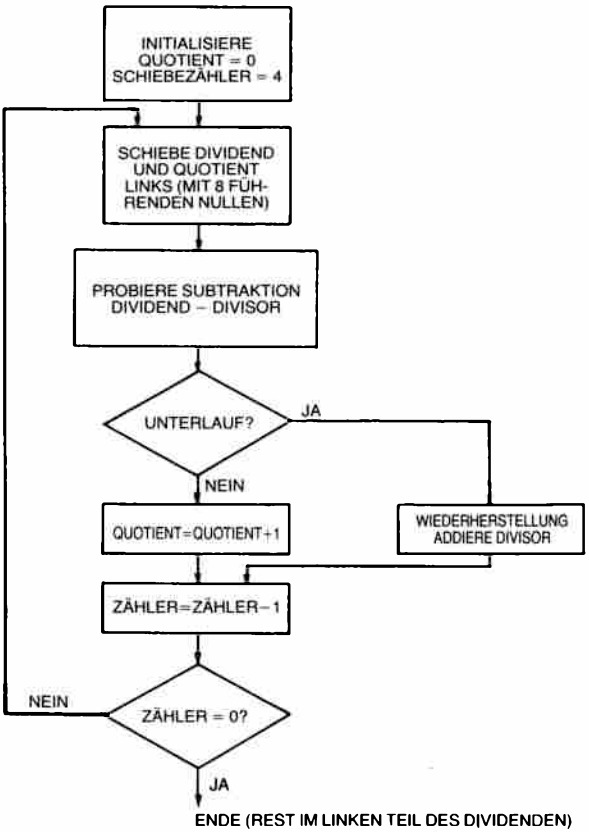


Abb. 3.30: Flußdiagramm für duale 8-Bit Division

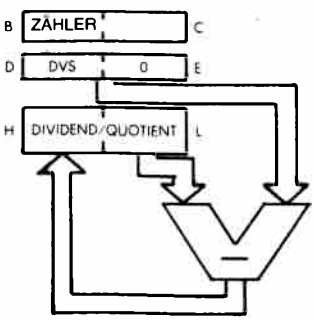


Abb. 3.31: 16/8 Bit Division – die Register

16/8 Bit Division

Als Beispiel wollen wir hier eine 16 durch 8 Bit Divison untersuchen, die einen 8-Bit-Quotienten und einen 8-Bit-Rest ergibt. Die Registerbelegung ist in Abb. 3.31 gezeigt.

Das Programm ist unten angegeben:

```
DIV168   LD    A,(DVSAD)   Lade Divisor
         LD    D,A         nach D
         LD    E,0
         LD    HL,(DVDAD)  Lade 16-Bit-Dividend
         LD    B,9         Initialisiere Zähler
         JP    REIN        Springe zu REIN
DIV      ADD   HL,HL       Schiebe Dividend links
REIN     XOR   A           Lösche Bit C
         SBC   HL,DE       Dividend – Divisor
         INC   HL          Quotient=Quotient+1
         JP    P,NOADD     Teste, ob Rest positiv
         ADD   HL,DE       Stelle wieder her,
                           wenn ötig
         DEC   HL          Quotient=Quotient+1
NOADD    DJNZ  DIV         Schleife, bis B=/
         RET
```

Abb. 3.32: 16/8 Bit Divisionsprogramm

Die ersten fünf Befehle des Programms laden den Divisor und den Dividenden in die vorgesehenen Register. Sie initialisieren außerdem den Zähler im Register B mit dem Wert 9. Beachten Sie wieder, daß das Register B ein bevorzugter Platz für einen Zähler ist, da man dann den speziellen Z80-Befehl DJNZ verwenden kann:

```
DIV168   LD    A,(DVSAD)
         LD    D,A
         LD    E,0
         LD    HL,(DVDAD)
         LD    B,9
```

Dann wird der Divisor von dem Dividenden subtrahiert. Da man den Befehl SBC verwenden muß (es gibt keine 16-Bit-Subtraktion ohne Übertragsbit), muß das Übertragsbit vorher auf den Wert „0" gesetzt werden. Dieses kann auf verschiedene Arten durchgeführt werden. Das Übertragsbit kann man löschen durch Operationen wie:

```
XOR   A
AND   A
OR    A
```

Hier wird ein XOR verwendet:

```
REIN XOR   A
```

Dann kann die Subtraktion ausgeführt werden:

```
        SBC   HL,DE
```

Es wird erwartet, daß die Subtraktion erfolgreich ist, d. h. daß der Rest positiv ist. Diesen Schritt nennt man „Subtraktion auf Probe" (beachten Sie dabei das Flußdiagramm in Abb. 3.30). Deshalb wird der Quotient um eins inkrementiert. Schlug die Subtraktion tatsächlich aber fehl (d. h. der Rest ist negativ), muß der Quotient später um eins dekrementiert werden:

```
        INC   HL
```

Dann wird das Ergebnis der Subtraktion getestet:

```
        JP    P,NOADD
```

Ist der Rest positiv oder null, war die Subtraktion erfolgreich, und es ist nicht nötig, sie rückgängig zu machen. Das Programm springt zur Adresse NOADD. Sonst muß der alte Wert des Dividenden wiederhergestellt werden, indem der Divisor wieder addiert wird, und der Quotient muß dekrementiert werden. Dies wird mit den nächsten Befehlen ausgeführt:

```
        ADD   HL,DE
        DEC   HL
```

Dann wird der Zähler B dekrementiert und auf den Wert „0" getestet. Solange B nicht Null ist, wird die Schleife weiter ausgeführt.

```
NOADD   DJNZ  DIV
        RET
```

Der resultierende Dividend wird links geschoben und der nächste Subtraktionsversuch erwartet.

```
DIV     ADD   HL,HL
```

Aufgabe 3.27: Überprüfen Sie die Arbeitsweise dieses Divisionsprogramms von Hand, indem sie die Tabelle in Abb. 3.33 ausfüllen, wie bei der Multiplikation in Aufgabe 3.18. Beachten Sie, daß der Inhalt von D nicht in Abb. 3.33 eingetragen werden muß, da er nirgends geändert wird.

MARKE	BEFEHL	B	H	L

Abb. 3.33: Formular für das Divisionsprogramm

8-Bit-Division

Das folgende Programm wendet ein wiederherstellendes Verfahren an und ergibt einen komplementierten Quotienten in A. Es dividiert 8 Bit durch 8 Bit (ohne Vorzeichen).

```
E ist der Dividend
C ist der Divisor
A ist der Quotient
B ist der Rest
DIV88   XOR   A       Lösche Akkumulator
        LD    B,8     Schleifenzähler
LOOP88  RL    E       Rotiere Übertragsbit in Akku
                      – Dividend
        RLA           Übertragsbit wird Null sein
        SUB   C       Versuche den Divisor zu
                      subtrahieren
        JR    NC,$+3  Subtraktion ok
        ADD   A,C     Stelle Akkumulator wieder
                      her, setze Übertragsbit
        DJNZ  LOOP88
        LD    B,A     Rest nach B
        LD    A,E     Quotient
        RLA           Schiebe das letzte Ergebnisbit
                      hinein
        CPL           Komplementiere die Bits
        RET
```

Beachten Sie: Das Symbol „$“ im sechsten Befehl stellt den Wert des Befehlszählers dar.

Nicht-wiederherstellende Division

Das folgende Programm führt eine ganzzahlige 16 Bit durch 15 Bit Division durch und wendet dabei ein nicht-wiederherstellendes Verfahren an. IX zeigt auf den Dividenden, IY auf den Divisor (von Null verschieden). Das Ergebnis steht dann in IX (siehe Abb. 3.34).

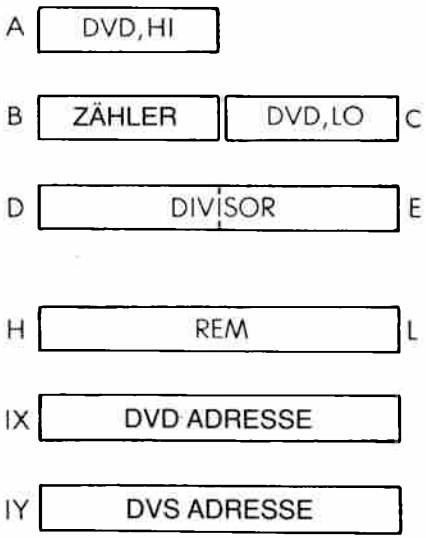


Abb. 3.34: Nicht-wiederherstellende Division – die Register

Register B wird als Zähler verwendet und anfangs auf 16 gesetzt.
A und C enthalten den Dividenden
D und E enthalten den Divisor
H und L enthalten das Ergebnis
Der 16-Bit-Dividend wird nach links geschoben durch:

```
RL  C
RLA
```

Der Rest wird nach links geschoben durch:

```
ADC  HL,HL
```

Der Quotient steht dann schließlich in B und C, der Rest in HL. Das Programm folgt.

```
DIV16    LD    B,(IX+1)
         LD    C,(IX)
         LD    D,(IY+1)
         LD    E,(IY)
         LD    A,D
         OR    E          Divisor oben oder Divisor unten
         JR    Z,ERROR    Test, ob Divisor Null
         LD    A,B        DVD oben
         LD    HL,0       Lösche Ergebnis
         LD    B.16D      Zähler
TRIALSB  RL    C          Rotiere Ergebnis und Akkumulator
                          links
         RLA
         ADC   HL,HL      Links Schieben, Übertragsbit wird nie
                          gesetzt
         SBC   HL,DE      Minus Divisor
NULL     CCF               Ergebnisbit
         JR    NC,NGV     Akkumulator negativ?
PTV      DJNZ  TRIALSB    Zähler Null?
         JP    DONE
RESTOR   RL    C          Rotiere Ergebnis und Akkumulator
                          links
         RLA
         ADC   HL,HL      Siehe oben
         AND   A
         ADC   HL,HL      Wiederherstellung durch Addition des
                          Divisors
         JR    C,PTV      Ergebnis positiv
         JR    Z,NULL     Ergebnis Null
NGV      DJNZ  RESTOR     Zähler Null?
DONE     RL    C          Schiebe Ergebnisbit ein
         RLA
         ADD   HL,DE      Korrigiere Rest
         LD    B,A        Quotient in BC
         RET
```

Aufgabe 3.28: Vergleiche das vorhergehende Programm mit dem folgenden, das ein wiederherstellendes Verfahren benutzt:

```
                Dividend in AC
                Divisor in DE
                Quotient in AC
                Rest in HL
DIV16     LD    HL,0        Lösche Akkumulator
          LD    B,16D       Setze Zähler
LOOP16    RL    C           Rotiere Akkumulator – Ergebnis links
          RLA
          ADC   HL,HL       Links Schieben
          SBC   HL,DE       Versuch, den Divisor zu subtrahieren
          JR    NC,$+3      Subtraktion war ok
          ADD   HL,DE       Stelle Akkumulator wieder her
          CCF               Berechne Ergebnisbit
          DJNZ  LOOP16      Zähler nicht Null
          RL    C           Schiebe letztes Ergebnisbit ein
          RET
```

Achtung: Das Symbol $ bedeutet „momentane Adresse" (im siebten Befehl).

Logische Operationen

Eine andere Klasse von Befehlen, die von der ALU im Mikroprozessor ausgeführt werden können, ist der Satz der *logischen Befehle.* Diese enthalten: AND, OR und exklusives OR (XOR). Zusätzlich kann man hier die Schiebe- und Rotieroperationen einschließen, die schon verwendet wurden, sowie die Vergleichsbefehle, die beim Z80 CP genannt werden. Der Gebrauch von AND, OR und XOR wird im Kapitel 4 bei dem Befehlssatz beschrieben.

Wir wollen jetzt ein kurzes Programm entwerfen, das überprüft, ob eine gegebene Speicherstelle, genannt LOC, den Wert „0", den Wert „1" oder etwas anderes enthält.

Das Programm führt die Vergleichsoperation ein und führt eine Reihe logischer Tests aus. Abhängig von dem Ergebnis des Vergleichs wird der eine oder der andere Programmteil ausgeführt.

Das Programm ist unten angegeben:

```
            LD    A,(LOC) Lies Zeichen in LOC
            CP    00H     Vergleich mit Null
            JP    Z,ZERO  Ist es eine 0?
            CP    01H     Vergleich mit Eins
            JP    Z,ONE
NONEFOUND   ...           Weder Null noch Eins gefunden
            ...
ZERO        ...
            ...
ONE         ...
```

Der erste Befehl „LD A,(LOC)" liest den Inhalt der Speicherstelle LOC und lädt ihn in den Akkumulator. Dies ist das Zeichen, das wir testen wollen. Mit dem folgenden Befehl wird es mit dem Wert Null verglichen:

```
CP  00H
```

Dieser Befehl vergleicht den Inhalt des Akkumulators mit dem hexadezimalen Wert „00", d. h. mit dem Bitmuster „0000 0000". Dieser Vergleichsbefehl setzt das Bit Z im Flagregister auf den Wert „1", wenn der Vergleich zutrifft. Dieses Bit wird von dem nächsten Befehl getestet:

```
JP  Z,ZERO
```

Der Sprungbefehl testet den Wert des Bits Z. Ist der Vergleich zutreffend, dann wird das Bit Z auf Eins gesetzt und der Sprung wird ausgführt. Das Programm springt dann zur Adresse ZERO. Trifft der Vergleich nicht zu, wird der nächste folgende Befehl ausgeführt:

```
CP  01H
```

Der folgende Sprungbefehl wird entsprechend zur Adresse ONE verzweigen, wenn der Vergleich zutrifft. Ist keiner der Vergleiche zutreffend, wird der Befehl an der Adresse NONEFOUND ausgeführt.

```
            JP  Z,ONE
NONEFOUND ...
```

Dieses Programm wurde vorgestellt, um die Nützlichkeit des Vergleichsbefehls, auf den ein Sprung folgt, zu zeigen. Diese Kombination wird in den folgenden Programmen oft verwendet werden.

Aufgabe 3.29: Verwende die Definition des Befehls LD A,(LOC) aus dem nächsten Kapitel. Untersuche die Wirkung dieses Befehls auf die Flags, falls diese beeinflußt werden. Ist der zweite Befehl des Programms notwendig (CP 00H)?

Aufgabe 3.30: Schreibe ein Programm, das den Inhalt der Speicherzelle „24" liest und zu einer „STERN" genannten Adresse verzweigt, wenn in der Speicherzelle 24 das Zeichen „*" stand. Die Bitfolge für das Zeichen „*" in Binärdarstellung ist „00101010".

Zusammenfassung der Befehle

Wir haben jetzt die meisten wichtigen Befehle des Z80 studiert, indem wir sie verwendet haben. Wir haben Werte zwischen dem Speicher und den Registern übertragen. Wir haben mit solchen Daten arithmetische und logische Operationen ausgeführt. Wir haben sie getestet und haben abhängig vom Ergebnis der Tests verschiedene Programmteile ausgeführt. Teilweise haben wir spezielle „automatische" Z80-Befehle wie DJNZ verwendet, um Programme zu verkürzen. Andere automatische Befehle wie LDDR, CPIR, INIR werden im weiteren eingeführt werden.

Spezielle Vorzüge des Z80 haben wir voll genutzt, um die Programme zu vereinfachen (z. B. die Befehle für 16-Bit-Register), und der Leser sollte darauf achten, diese Programme nicht auf einem 8080 zu verwenden: sie wurden für den Z80 optimiert.

Wir haben auch eine Programmstruktur, eine sogen. Schleife, eingeführt. Eine andere wichtige Struktur wird jetzt vorgestellt: das Unterprogramm.

Unterprogramme

Im Prinzip ist ein Unterprogramm einfach ein Block von Befehlen, dem der Programmierer einen Namen gegeben hat. In der Praxis muß ein Unterprogramm mit einem speziellen Befehl beginnen, *Unterprogramm-Eröffnung* genannt, der es für den Assembler als Unterprogramm identifiziert. Es muß auch mit einem speziellen Befehl enden, der *Return* (Rücksprung) genannt wird. Wir wollen zuerst die Verwendung eines Unterprogramms veranschaulichen, um seine Bedeutung zu zeigen. Dann werden wir untersuchen, wie es tatsächlich ausgeführt wird.

Der Gebrauch eines Unterprogramms ist in Abb. 3.35 veranschaulicht. Das Hauptprogramm erscheint auf der linken Seite der Abbildung. Rechts ist das Unterprogramm symbolisch dargestellt.

Wir wollen nun den Mechanismus des Unterprogramms untersuchen. Die Zeilen des Hauptprogramms werden nacheinander abgearbeitet, bis ein neuer Befehl auftritt: CALL SUB. Dieser Befehl ist der *Unterprogrammaufruf* und bewirkt einen Sprung zum Unterprogramm. Dies

Der Nachteil eines Unterprogramms sollte schon klar werden, wenn man nur den Ablauf der Ausführung von Hauptprogramm und Unterprogramm verfolgt. Ein Unterprogramm führt zu einer verlangsamten Ausführung, weil zusätzliche Befehle abgearbeitet werden müssen: CALL SUB und RETURN.

Ablauf des Unterprogramm-Mechanismus

Wir wollen hier untersuchen, wie die beiden Befehle CALL SUB und RETURN innerhalb des Mikroprozessors ausgeführt werden. Die Folge des Befehls CALL SUB ist es, daß der nächste Befehl von einer neuen Adresse geholt wird. Sie werden sich erinnern (oder sonst im Kapitel 1 nachlesen), daß die Adresse des Befehls, der in einem Computer als nächster ausgeführt werden soll, im Befehlszähler (PC) steht. Dies heißt, der Befehl CALL SUB bewirkt, daß in das Register PC ein neuer Inhalt geladen wird. Er bewirkt, daß die Anfangsadresse des Unterprogramms in den Befehlszähler geladen wird. *Reicht das aber wirklich aus?*

Um diese Frage zu beantworten, wollen wir den anderen Befehl betrachten, der eingebaut werden muß: RETURN. Wie der Name sagt, muß der Befehl RETURN einen Rücksprung zu dem Befehl bewirken, der auf den CALL SUB folgt. Dies ist aber nur möglich, wenn diese Adresse irgendwo gespeichert wurde. Diese Adresse war der Inhalt des Befehlszählers, als der Befehl CALL SUB auftrat. Dies ist der Fall, weil der Befehlszähler immer dann automatisch inkrementiert wird, wenn er benutzt wurde (siehe Kapitel 1). Genau das ist die Adresse, die wir aufbewahren wollen, so daß wir später den RETURN ausführen können.

Das nächste Problem ist: Wo können wir die Rücksprungadresse aufbewahren? Diese Adresse muß an einer Stelle gespeichert werden, wo sichergestellt ist, daß sie nicht gelöscht wird.

Wir wollen jedoch die folgende Situation betrachten, die in Abb. 3.36 veranschaulicht ist. In diesem Beispiel enthält das Unterprogramm 1 einen Call (Aufruf) des Unterprogramms 2. Unser Verfahren sollte auch in diesem Fall funktionieren. Natürlich kann es auch mehr als zwei Unterprogramme mit allgemein N „verschachtelten" Aufrufen geben. Deshalb muß das Verfahren den Befehlszähler immer wieder abspeichern, wenn ein neuer CALL erreicht wird. Dies setzt voraus, daß uns wenigstens 2N Speicherplätze für dieses Verfahren zur Verfügung stehen. Außerdem müssen wir zuerst aus dem Unterprogramm SUB2 und danach aus dem Unterprogramm SUB1 zurückspringen. Mit anderen Worten, wir brauchen eine Struktur, die die zeitliche Reihenfolge aufbewahrt, in der die Adressen gespeichert wurden.

Diese Struktur hat einen Namen, und sie wurde bereits eingeführt. Es ist der *Stapel*. Abb. 3.38 zeigt den jeweiligen Inhalt des Stapels während aufeinanderfolgender Unterprogrammaufrufe. Wir wollen uns zuerst das Hauptprogramm anschauen. Bei der Adresse 100 wird der erste CALL erreicht: CALL SUB1. Wir wollen annehmen, daß der Unter-

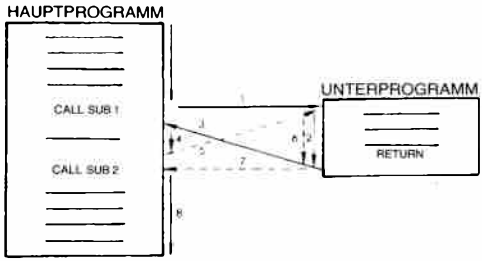


Abb. 3.35: Unterprogrammaufrufe

heißt, daß nach dem Befehl CALL SUB der erste Befehl innerhalb des Unterprogramms ausgeführt wird. Dies macht der Pfeil Nr. 1 in Abb. 3.35 deutlich.

Dann wird das Unterprogramm genau wie jedes andere Programm auch ausgeführt. Wir wollen annehmen, daß das Unterprogramm keine Calls enthält. Der letzte Befehl dieses Unterprogramms ist ein RETURN. Dies ist ein Befehl, der einen Rücksprung ins Hauptprogramm bewirkt. Der Befehl, der nach dem RETURN als nächster ausgeführt wird, ist der Befehl, der im Hauptprogramm dem Befehl CALL SUB folgt. Dies verdeutlicht der Pfeil Nr. 3 in der Abb. 3.35. Das Programm wird dann weiter ausgeführt, wie es der Pfeil Nr. 4 anzeigt.

Innerhalb des Hauptprogramms erscheint dann ein zweiter Aufruf CALL SUB. Es wird wieder ein Sprung ausgeführt, wie es der Pfeil Nr. 5 zeigt. Dies heißt, daß nach dem zweiten Befehl CALL SUB wiederum das Unterprogramm ausgeführt wird.

Wann immer innerhalb des Unterprogramms der Befehl RETURN erreicht wird, tritt ein Rücksprung zu dem nächsten Befehl nach den betreffenden CALL SUB auf. Dies ist mit Pfeil Nr. 7 veranschaulicht. Nach der Rückkehr ins Hauptprogramm wird dort die Ausführung normal fortgesetzt, wie es der Pfeil Nr. 8 zeigt.

Die Wirkungsweise der beiden speziellen Befehle CALL SUB und RETURN sollte jetzt klar sein. Was ist der Vorteil des Unterprogrammverfahrens.

Der wesentliche Vorzug eines Unterprogramms ist es, daß es von beliebig vielen Stellen im Hauptprogramm aus aufgerufen und wiederholt ausgeführt werden kann, ohne daß man es mehrfach schreiben muß. Ein erster Vorteil ist, daß dieses Verfahren Speicherplatz spart, weil man das Unterprogramm nicht jedesmal neu einfügen muß. Ein zweiter Vorteil ist, daß der Programmierer ein spezielles Unterprogramm nur einmal entwerfen braucht, und es dann wiederholt verwenden kann. Dies ist eine wesentliche Vereinfachung für den Programmentwurf.

Aufgabe 3.31: Was ist der hauptsächliche Nachteil eines Unterprogramms? (die Antwort folgt.)

programmaufruf in diesem Mikroprozessor drei Byte belegt (RST ist eine Ausnahme davon). Die nächste folgende Adresse ist deshalb nicht „101" sondern „103". Der Befehl CALL belegt die Adressen „100", „101" und „102". Da das Steuerwerk des Z80 „weiß", daß dies ein Dreibytebefehl ist, hat der Befehlszähler den Inhalt „103", sobald der Befehl dekodiert ist. Die Wirkung des Call ist es, den Wert „280" in den Befehlszähler zu laden. „280" ist die Startadresse von SUB1.

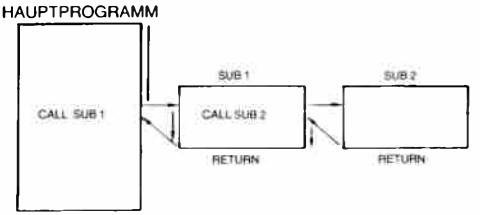


Abb. 3.36: Verschachtelte Aufrufe

Jetzt sind wir in der Lage, die Wirkung des Befehls RETURN und die korrekte Arbeitsweise unseres Stapelverfahrens zu demonstrieren. Die Ausführung wird innerhalb von SUB2 fortgesetzt, bis zum Zeitpunkt Time3 der Befehl RETURN erreicht wird. Die Wirkungsweise des Befehls RETURN besteht darin, das oberste Element des Stapels in den Befehlszähler zu holen (Pop). Mit anderen Worten, der Befehlszähler wird auf seinen Wert vor dem Eintritt in das Unterprogramm zurückgesetzt. Das oberste Stapelelement ist in unserem Beispiel „303". Abb. 3.38 zeigt, daß zum Zeitpunkt Time3 der Wert „303" vom Stapel entfernt und in den Befehlszähler zurückgeholt wurde. Deshalb wird die Befehlsausführung bei der Adresse „303" fortgesetzt. Zum Zeitpunkt Time4 wird der RETURN von SUB1 erreicht. Das oberste Element des Stapels enthält den Wert „103". Es wird vom Stapel geholt (Pop) und in den Befehlszähler gebracht. Deshalb wird die Ausführung bei der Adresse „103" innerhalb des Hauptprogramms fortgesetzt. Dies ist in der Tat die gewünschte Arbeitsweise. Abb. 3.38 zeigt, daß der Stapel zum Zeitpunkt Time4 wieder leer ist. Das Verfahren funktioniert.

Das Verfahren des Unterprogrammaufrufs funktioniert bis zur maximalen Länge des Stapels. Deshalb waren frühere Mikroprozessoren, die einen Stapel aus vier oder acht Registern hatten, prinzipiell auf vier oder acht Unterprogrammebenen beschränkt.

Beachten Sie, daß die Unterprogramme in den Abbildungen 3.36 und 3.37 rechts vom Hauptprogramm gezeichnet sind. Dies geschah nur wegen der Übersichtlichkeit des Programms. In Wirklichkeit gibt der Benutzer die Unterprogramme als normale Befehle des Programms ein. Wenn man die Liste des vollständigen Programms auf einem Blatt Papier zusammenstellt, können die Unterprogramme am Anfang des

Texts, mittendrin oder am Ende stehen. Deshalb wird ihnen eine Unterprogrammeröffnung vorangestellt: sie müssen identifiziert werden. Die speziellen Befehle sagen dem Assembler, daß das Folgende als Unterprogramm behandelt werden soll. Solche *Assemblerdirektiven* werden in Kapitel 10 diskutiert.

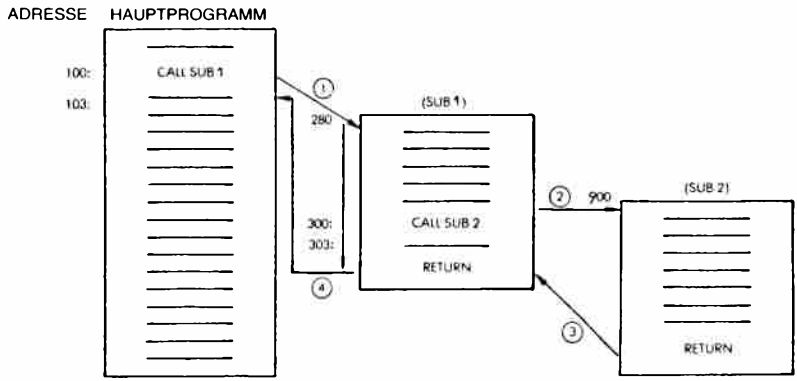


Abb. 3.37: Die Unterprogrammaufrufe

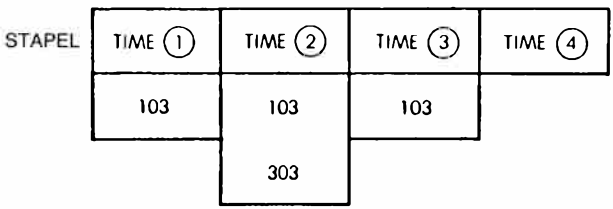


Abb. 3.38: Der Stapel zu verschiedenen Zeitpunkten

Unterprogramme beim Z80

Die Grundkonzepte von Unterprogrammen wurden jetzt vorgestellt. Es wurde gezeigt, daß zur Ausführung dieses Verfahrens ein Stapel nötig ist. Der Z80 ist mit einem 16-Bit-Stapelanzeiger ausgestattet. Der Stapel kann deshalb an einer beliebigen Stelle im Speicher stehen und er kann bis zu 64k Byte lang sein (1k = 1024), wenn wir annehmen, daß sie für diesen Zweck zur Verfügung stehen. In der Praxis wird der Programmierer die Anfangsadresse des Stapels und seine maximale Ausdehnung festlegen, bevor er das Programm schreibt. Dann ist ein Speicherbereich für den Stapel reserviert.

Der Befehl zum Unterprogrammaufruf heißt beim Z80 CALL und kommt in zwei Versionen vor: den direkten oder unbedingten Aufruf wie CALL ADRESSE haben wir schon beschrieben. Zusätzlich ist der Z80 mit bedingten Aufrufen ausgestattet, die ein Unterprogramm aufrufen, falls eine Bedingung erfüllt ist. Beispiel: CALL NZ,SUB1 ruft dann das Unterprogramm SUB1 auf, wenn das Ergebnis der vorhergehenden Operationen nicht Null war. Dies ist eine nützliche Möglichkeit, da viele Unterprogrammaufrufe bedingt sind, d. h. nur dann auftreten, wenn bestimmte Bedingungen erfüllt sind.

CALL CC,NN wird nur dann ausgeführt, wenn die Bedingung, die durch „CC" festgelegt wird, erfüllt ist. CC ist ein Satz von drei Bit (Bit 4, 5 und 6 des Opcodes), die bis zu acht Bedingungen festlegen können. Diese entsprechen den vier Flags „Z", „C", „P/U" und „S", die entweder Null oder nicht Null sein können.

Entsprechend gibt es zwei Arten von Return-Befehlen: RET und RET CC.

RET ist der grundsätzliche Rücksprungbefehl. Er belegt nur ein Byte und bewirkt, daß die oberen beiden Byte des Stapels wieder in den Befehlszähler geladen werden. Er ist nicht bedingt.

RET CC hat die gleiche Wirkung, mit der Ausnahme, daß er nur ausgeführt wird, wenn die durch CC festgelegte Bedingung erfüllt ist. Die Bedingungsbits sind die gleichen, wie bei dem eben beschriebenen Befehl CALL CC.

Zusätzlich gibt es zwei spezielle Arten von Rücksprungbefehlen, mit denen Interruptroutinen abgeschlossen werden: RETI und RETN. Sie werden in dem Kapitel über Z80-Befehle und in dem Kapitel über Interrupts beschrieben.

Schließlich gibt es noch einen speziellen Befehl, der einem Unterprogrammaufruf entspricht, der aber Verzweigungen nur zu einer von acht Adressen in der Seite Null zuläßt. Dies ist der Befehl RST P. Er ist ein Einbytebefehl, der automatisch den Befehlszähler auf dem Stapel ablegt und eine Verzweigung zu der Dreibitadresse ausführt, die im Feld P festgelegt ist. Das Feld P entspricht den Bits 4, 5 und 6 des Befehls, multipliziert mit acht.

Mit anderen Worten, wenn die Bits 4, 5 und 6 „000" sind, wird die Adresse 00H angesprungen. Sind diese Bits „001", wird zur Adresse 08H verzweigt, usw. bis zu 111, das einen Sprung zur Adresse 38H bewirkt. Der Befehl RST ist in bezug auf die Geschwindigkeit sehr effizient, da er ein Einbytebefehl ist. Er kann jedoch nur acht Adressen in der Seite Null anspringen. Zusätzlich liegen diese Adressen nur acht Byte auseinander. Dieser Befehl wurde vom 8080 übernommen und wurde dort weitgehend für Interrupts verwendet. Dies wird im Kapitel über Interrupts beschrieben werden. Der Programmierer kann diesen Befehl jedoch für beliebige andere Zwecke benutzen und er sollte als möglicher spezieller Unterprogrammaufruf betrachtet werden.

Beispiele für Unterprogramme

Die meisten Programme, die wir entwickelt haben und die wir noch entwickeln werden, würden normalerweise als Unterprogramm entwickelt. Das Multiplikationsprogramm zum Beispiel wird normalerweise von vielen Programmbereichen verwendet. Um die Programmentwicklung zu vereinfachen und übersichtlicher zu gestalten, ist es daher angebracht, ein Unterprogramm zu definieren, das beispielsweise den Namen MULT erhält. Am Ende dieses Unterprogramms würden wir einfach den Befehl RET anfügen.

Aufgabe 3.32: Wenn „MULT" als Unterprogramm verwendet wird, zerstört es dann irgendwelche internen Flags oder Register?

Rekursion

Die Bezeichnung Rekursion wird verwendet, um anzuzeigen, daß sich ein Programm selbst aufruft. Wenn Sie das Konzept des Unterprogramms verstanden haben, sollten Sie folgende Frage beantworten können.

Aufgabe 3.33: Ist es zulässig, daß sich ein Unterprogramm selbst aufruft? (Mit anderen Worten: funktioniert alles, auch wenn sich ein Unterprogramm selbst aufruft?) Wenn Sie nicht sicher sind, zeichnen Sie den Stapel und füllen Sie ihn der Reihe nach mit Adressen. Betrachten Sie dann Register und Speicher (siehe Abb. 3.18) und entscheiden Sie, ob ein Problem auftritt.

Interrupts werden im Kapitel Ein-/Ausgabe (Kapitel 6) diskutiert. Alle Rücksprünge sind Einbytebefehle, alle Unterprogrammaufrufe sind Dreibytebefehle (außer RST).

Aufgabe 3.34: Schlagen Sie im nächsten Kapitel die Ausführungszeiten von den Befehlen CALL und RET nach. Warum ist der Rücksprung von einem Unterprogramm um so viel schneller als der Aufruf? (Hinweis: Wenn die Antwort nicht klar ist, dann betrachten Sie nochmals das Unterprogrammverfahren, und analysieren Sie, welche internen Schritte ausgeführt werden müssen.)

Unterprogrammparameter

Wenn ein Unterprogramm aufgerufen wird, dann erwartet man normalerweise, daß das Unterprogramm irgendwelche Daten bearbeitet. Im Fall der Multiplikation will man beispielsweise zwei Zahlen an das Unterprogramm übergeben, die multipliziert werden sollen. Bei dem Multiplikationsprogramm sahen wir, daß dieses Unterprogramm den Multiplikator und den Multiplikanden in bestimmten Speicherzellen erwartete. Dies veranschaulicht eine Methode der Parameterübergabe: im Speicher. Zwei andere Techniken werden noch verwendet, so daß wir drei Arten der Parameterübergabe haben:

1 – in Registern
2 – im Speicher
3 – im Stapel

Register können zur Parameterübergabe dienen. Wenn die Register frei sind, ist dies eine günstige Methode, da man keine Speicheradresse festlegen muß: Das Unterprogramm bleibt unabhängig vom Speicher verwendbar. Wird eine feste Speicheradresse verwendet, so muß jeder andere Benutzer des Unterprogramms sorgfältig darauf achten, daß er die gleiche Vereinbarung verwendet und daß die Speicherstelle auch frei ist (beachten Sie Aufgabe 3.19 oben). Deshalb wird in vielen Fällen einfach ein Speicherblock reserviert, in dem die Parameter an die verschiedenen Unterprogramme übergeben werden.

Die Verwendung des Speichers hat den Vorteil größerer Flexibilität (mehr Daten), bewirkt aber eine geringere Universalität und legt das Unterprogramm auf einen vorgegebenen Speicherbereich fest.

Die Parameter *im Stapel* abzulegen, hat den gleichen Vorteil wie die Verwendung der Register: man ist vom Speicher unabhängig. Das Unterprogramm weiß einfach, daß ihm sagen wir zwei Parameter übergeben werden, die oben auf dem Stapel liegen. Natürlich hat dies Nachteile: Es bringt im Stapel hintereinander Daten und Adressen und vermindert deshalb die zulässige Anzahl von Unterprogrammebenen. Auch macht es die Verwendung des Stapels erheblich komplizierter, und es kann mehrere Stapel erforderlich machen.

Die Auswahl liegt beim Programmierer. Im allgemeinen will man so lange wie möglich von tatsächlichen Speicheradressen unabhängig bleiben.

Stehen keine Register zur Verfügung, dann ist der Stapel eine mögliche Lösung. Wenn jedoch eine große Menge Information an das Unterprogramm übergeben werden muß, dann wird diese Information direkt im Speicher stehen müssen. Eine elegante Methode, einen Block von Daten zu übergeben, ist es, einfach einen *Zeiger* auf die Information zu übergeben. Ein Zeiger ist die Adresse des Blockanfangs. Ein Zeiger kann in einem Register, auf dem Stapel (zum Speichern einer 16-Bit-Adresse kann man zwei Plätze im Stapel verwenden) oder in einer festgelegten Speichzerzelle übergeben werden.

Ist schließlich keine der beiden Möglichkeiten anwendbar, kann man mit dem Unterprogramm vereinbaren, daß die Daten bei irgendeiner festen Speicheradresse stehen (dem „Briefkasten").

Aufgabe 3.35: Welche der drei Methoden ist zur Rekursion am besten geeignet?

Unterprogrammbibliothek

Es gibt einen entscheidenden Vorteil, Programme in unterscheidbare Unterprogramme zu strukturieren: Sie können unabhängig voneinander korrigiert werden, und sie können einen Namen haben. Vorausgesetzt,

daß sie in getrennten Bereichen des Programms stehen, kann man sie herausnehmen und so eine Bibliothek nützlicher Unterprogramme aufbauen. Allerdings gibt es bei der Computerprogrammierung keine universellen Heilmittel. Die systematische Verwendung von Unterprogrammen, die nach ihrer Funktion zusammengefaßt sind, kann auch zu schlechter Effektivität führen.

Der geschickte Programmierer muß die Vorteile gegen die Nachteile abwägen.

Zusammenfassung

In diesem Kapitel wurde dargestellt, auf welche Art Information innerhalb des Z80 durch Befehle verarbeitet wird. Algorithmen von wachsender Komplexität wurden eingeführt und in Programme übersetzt. Die hauptsächlichen Arten von Befehlen wurden verwendet und erklärt.

Wichtige Strukturen wie Schleifen, Stapel und Unterprogramme wurden definiert.

Sie sollten jetzt ein grundlegendes Verständnis vom Programmieren und von den wichtigeren Techniken, die man in Standardanwendungen benutzt, erworben haben. Jetzt wollen wir die verfügbaren Befehle studieren.

```
          A=00 BC=0000 DE=0000 HL=0000 S=0300 P=0100 0100' LD   BC,(0200)
          A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00      (0200')
          A=00 BC=0003 DE=0000 HL=0000 S=0300 P=0104 0104' LD   B,08
          A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
          A=00 BC=0803 DE=0000 HL=0000 S=0300 P=0106 0106' LD   DE,(0202)
          A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00      (0202')
          A=00 BC=0803 DE=0005 HL=0000 S=0300 P=010A 010A' LD   D,00
          A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
          A=00 BC=0803 DE=0005 HL=0000 S=0300 P=010C 010C' LD   HL,0000
          A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00      (0000')
          A=00 BC=0803 DE=0005 HL=0000 S=0300 P=010F 010F' SRL  C
          A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
        C A=00 BC=0801 DE=0005 HL=0000 S=0300 P=0111 0111' JR   NC,0114
          A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00      (0114')
        C A=00 BC=0801 DE=0005 HL=0000 S=0300 P=0113 0113' ADD  HL,DE
          A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
          A=00 BC=0801 DE=0005 HL=0005 S=0300 P=0114 0114' SLA  E
          A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
      V   A=00 BC=0801 DE=000A HL=0005 S=0300 P=0116 0116' RL   D
          A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
  Z V     A=00 BC=0801 DE=000A HL=0005 S=0300 P=0118 0118' DEC  B
          A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
      N   A=00 BC=0701 DE=000A HL=0005 S=0300 P=0119 0119' JP   NZ,010F
          A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00      (010F')
      N   A=00 BC=0701 DE=000A HL=0005 S=0300 P=010F 010F' SRL  C
          A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
  Z V C   A=00 BC=0700 DE=000A HL=0005 S=0300 P=0111 0111' JR   NC,0114
          A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00      (0114')
  Z V C   A=00 BC=0700 DE=000A HL=0005 S=0300 P=0113 0113' ADD  HL,DE
          A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
  Z V     A=00 BC=0700 DE=000A HL=000F S=0300 P=0114 0114' SLA  E
          A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
      V   A=00 BC=0700 DE=0014 HL=000F S=0300 P=0116 0116' RL   D
          A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
  Z V     A=00 BC=0700 DE=0014 HL=000F S=0300 P=0118 0118' DEC  B
          A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
      N   A=00 BC=0600 DE=0014 HL=000F S=0300 P=0119 0119' JP   NZ,010F
          A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00      (010F')
      N   A=00 BC=0600 DE=0014 HL=000F S=0300 P=010F 010F' SRL  C
          A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
  Z V     A=00 BC=0600 DE=0014 HL=000F S=0300 P=0111 0111' JR   NC,0114
          A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00      (0114')
  Z V     A=00 BC=0600 DE=0014 HL=000F S=0300 P=0114 0114' SLA  E
          A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
      V   A=00 BC=0600 DE=0028 HL=000F S=0300 P=0116 0116' RL   D
          A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
  Z V     A=00 BC=0600 DE=0028 HL=000F S=0300 P=0118 0118' DEC  B
          A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
      N   A=00 BC=0500 DE=0028 HL=000F S=0300 P=0119 0119' JP   NZ,010F
          A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00      (010F')
      N   A=00 BC=0500 DE=0028 HL=000F S=0300 P=010F 010F' SRL  C
          A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
  Z V     A=00 BC=0500 DE=0028 HL=000F S=0300 P=0111 0111' JR   NC,0114
          A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00      (0114')
  Z V     A=00 BC=0500 DE=0028 HL=000F S=0300 P=0114 0114' SLA  E
          A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
      V   A=00 BC=0500 DE=0050 HL=000F S=0300 P=0116 0116' RL   D
          A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
  Z V     A=00 BC=0500 DE=0050 HL=000F S=0300 P=0118 0118' DEC  B
          A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
      N   A=00 BC=0400 DE=0050 HL=000F S=0300 P=0119 0119' JP   NZ,010F
          A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00      (010F')
      N   A=00 BC=0400 DE=0050 HL=000F S=0300 P=010F 010F' SRL  C
          A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
```

Abb. 3.39: Multiplikation: eine vollständige Aufzeichnung

```
 Z  V      A=00 BC=0400 DE=0050 HL=000F S=0300 P=0111 0111' JR    NC,0114
           A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00      (0114')
 Z  V      A=00 BC=0400 DE=0050 HL=000F S=0300 P=0114 0114  SLA E
           A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
S   V      A=00 BC=0400 DE=00A0 HL=000F S=0300 P=0116 0116' RL  D
           A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
 Z  V      A=00 BC=0400 DE=00A0 HL=000F S=0300 P=0118 0118' DEC B
           A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
    N      A=00 BC=0300 DE=00A0 HL=000F S=0300 P=0119 0119' JP  NZ,010F
           A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00      (010F')
    N      A=00 BC=0300 DE=00A0 HL=000F S=0300 P=010F 010F' SRL C
           A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
 Z  V      A=00 BC=0300 DE=00A0 HL=000F S=0300 P=0111 0111' JR    NC,0114
           A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00      (0114')
 Z  V      A=00 BC=0300 DE=00A0 HL=000F S=0300 P=0114 0114  SLA E
           A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
      C    A=00 BC=0300 DE=0040 HL=000F S=0300 P=0116 0116' RL  D
           A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
           A=00 BC=0300 DE=0140 HL=000F S=0300 P=0118 0118' DEC B
           A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
    N      A=00 BC=0200 DE=0140 HL=000F S=0300 P=0119 0119' JP  NZ,010F
           A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00      (010F')
    N      A=00 BC=0200 DE=0140 HL=000F S=0300 P=010F 010F' SRL C
           A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
 Z  V      A=00 BC=0200 DE=0140 HL=000F S=0300 P=0111 0111' JR    NC,0114
           A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00      (0114')
 Z  V      A=00 BC=0200 DE=0140 HL=000F S=0300 P=0114 0114  SLA E
           A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
S          A=00 BC=0200 DE=0180 HL=000F S=0300 P=0116 0116' RL  D
           A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
           A=00 BC=0200 DE=0280 HL=000F S=0300 P=0118 0118' DEC B
           A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
    N      A=00 BC=0100 DE=0280 HL=000F S=0300 P=0119 0119' JP  NZ,010F
           A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00      (010F')
    N      A=00 BC=0100 DE=0280 HL=000F S=0300 P=010F 010F' SRL C
           A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
 Z  V      A=00 BC=0100 DE=0280 HL=000F S=0300 P=0111 0111' JR    NC,0114
           A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00      (0114')
 Z  V      A=00 BC=0100 DE=0280 HL=000F S=0300 P=0114 0114  SLA E
           A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
 Z  V  C   A=00 BC=0100 DE=0200 HL=000F S=0300 P=0116 0116' RL  D
           A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
    V      A=00 BC=0100 DE=0500 HL=000F S=0300 P=0118 0118' DEC B
           A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
 Z  N      A=00 BC=0000 DE=0500 HL=000F S=0300 P=0119 0119' JP  NZ,010F
           A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00      (010F')
 Z  N      A=00 BC=0000 DE=0500 HL=000F S=0300 P=011C 011C' LD  (0204),HL
           A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00      (0204')
 Z  N      A=00 BC=0000 DE=0500 HL=000F S=0300 P=011F 011F' NOP
           A'=00 B'=0000 D'=0000 H'=0000 X=0000 Y=0000 I=00
```

Abb. 3.39: Multiplikation: eine vollständige Aufzeichnung (Fortsetzung)

Antworten zu Aufgabe 3.18 (Multiplikation):

```
 CROMEMCO CDOS Z80 ASSEMBLER version 02.15                    PAGE 0001

0000'                0001          ORG     0100H
      (0200)         0002 MPRAD    DL      0200H
      (0202)         0003 MPDAD    DL      0202H
      (0204)         0004 RESAD    DL      0204H
                     0005 ;
0100  ED4B0002       0006 MPY88    LD      BC,(MPRAD)      ;LOAD MULTIPLIER INTO C
0104  0608           0007          LD      B,8             ;B IS BIT COUNTER
0106  ED5B0202       0008          LD      DE,(MPDAD)      ;LOAD MUTIPLICAND INTO E
010A  1600           0009          LD      D,0             ;CLEAR D
010C  210000         0010          LD      HL,0            ;SET RESULT TO 0
010F  CB39           0011 MULT     SRL     C               ;SHIFT MULTIPLIER BIT INTO CARRY
0111  3001           0012          JR      NC,NOADD        ;TEST CARRY
0113  19             0013          ADD     HL,DE           ;ADD MPD TO RESULT
0114  CB23           0014 NOADD    SLA     E               ;SHIFT MPD LEFT
0116  CB12           0015          RL      D               ;SAVE BIT IN D
0118  05             0016          DEC     B               ;DECREMENT SHIFT COUNTER
0119  C20F01         0017          JP      NZ,MULT         ;DO IT AGAIN IF COUNTER <> 0
011C  220402         0018          LD      (RESAD),HL      ;STORE RESULT
011F  (0000)         0019          END

Errors               0
```

Abb. 3.40: Das Multiplikationsprogramm (hexadezimal)

LABEL	INSTRUCTION	B	C	C (CARRY)	D	E	H	L
		00	00	0	00	00	00	00
MPY88	LD BC,(0200)	00	03	0	00	00	00	00
	LD B, 08	08	03	0	00	00	00	00
	LD DE,(0202)	08	03	0	00	05	00	00
	LD D, 00	08	03	0	00	05	00	00
	LD HL,0000	08	03	0	00	05	00	00
MULT	SRL C	08	01	1	00	05	00	00
	JR NC,0114	08	01	1	00	05	00	00
	ADD HL,DE	08	01	0	00	05	00	05
NOADD	SLA E	08	01	0	00	0A	00	05
	RL D	08	01	0	00	0A	00	05
	DEC B	07	01	0	00	0A	00	05
	JP NZ,010F	07	01	0	00	0A	00	05
MULT	SRL C	07	00	1	00	0A	00	05
	JR NC,0114	07	00	1	00	0A	00	05
	ADD HL,DE	07	00	0	00	0A	00	0F
NOADD	SLA E	07	00	0	00	14	00	0F
	RL D	07	00	0	00	14	00	0F
	DEC B	06	00	0	00	14	00	0F
	JP NZ,010F	06	00	0	00	14	00	0F

Abb. 3.41: Zwei Durchläufe durch die Schleife

4

Der Befehlssatz des Z80

Einführung

In diesem Kapitel werden wir zunächst die verschiedenen Klassen von Befehlen untersuchen, die in einem universellen Computer verfügbar sein sollten. Danach werden nacheinander alle Befehle, die im Z80 verfügbar sind, analysiert, und ihre Wirkung und die Art und Weise, wie sie Flags beeinflussen oder wie sie mit verschiedenen Adressierungsarten verwendet werden können, erklärt. Eine eingehende Diskussion von Adressierungstechniken folgt dann in Kapitel 5.

Klassen von Befehlen

Befehle kann man nach verschiedenen Gesichtspunkten sortieren und es gibt dafür keine Norm. Wir werden hier fünf Hauptgruppen von Befehlen unterscheiden:

1 – Transfer von Daten
2 – Bearbeitung von Daten
3 – Tests und Sprünge
4 – Eingabe und Ausgabe
5 – Steuerbefehle

Wir wollen jetzt nacheinander jede dieser Gruppen besprechen.

Transfer von Daten

Transferbefehle übertragen Daten zwischen Registern, zwischen einem Register und dem Speicher oder zwischen einem Register und einem Ein-/Ausgabe-Gerät. Für Register mit spezieller Funktion können spezielle Transferbefehle existieren. Beispielsweise sind Befehle wie Push und Pop für eine wirkungsvolle Arbeit mit dem Stack vorgesehen. Mit einem einzigen Befehl übertragen sie ein Datenwort zwischen Stack und Akku und ändern automatisch den Stackpointer.

Bearbeitung von Daten

Die Befehle zur Bearbeitung von Daten sind in fünf Klassen eingeteilt:

1 – Arithmetische Operationen (wie Plus/Minus)
2 – Manipulation einzelner Bits (Set und Reset)
3 – Inkrementieren und Dekrementieren
4 – logische Operationen (wie AND, OR, exklusives OR)
5 – Befehle zum Vertauschen und Schieben (wie Shift, Rotate)

Man sollte beachten, daß zur rationellen Bearbeitung von Daten leistungsfähige arithmetische Befehle wie Multiplikation und Division, von Nutzen sind. Leider sind diese auf den meisten Mikroprozessoren nicht verfügbar. Wünschenswert sind außerdem leistungsfähige Befehle zum Schieben und Vertauschen, wie Schieben von n Bit oder Vertauschen der Nibble, wobei die linke und die rechte Hälfte eines Bytes vertauscht werden. Auf den meisten Mikroprozessoren sind auch diese üblicherweise nicht verfügbar.

Bevor wir die eigentlichen Befehle des Z80 untersuchen, wollen wir den Unterschied zwischen *Schieben* und *Rotieren* diskutieren. Beim Schieben wird der Inhalt eines Registers oder einer Speicherzelle um ein Bit nach links oder nach rechts verschoben. Das Bit, das aus dem Register herausfällt, kommt in das Übertragsbit (Carrybit). Das Bit, das auf der anderen Seite hereinkommt, ist eine „0", außer beim „Arithmetischen-Rechts-Schieben", bei dem das MSB dupliziert wird.

Beim Rotieren gelangt das Bit, das herauskommt, ebenfalls ins Carry. Das Bit, das neu hereinkommt, ist der alte Inhalt des Carry. Dies entspricht einem 9-Bit-Rotieren. Oft wünscht man sich auch ein echtes 8-Bit-Rotieren, bei dem das Bit, das auf der einen Seite herausfällt, auf der

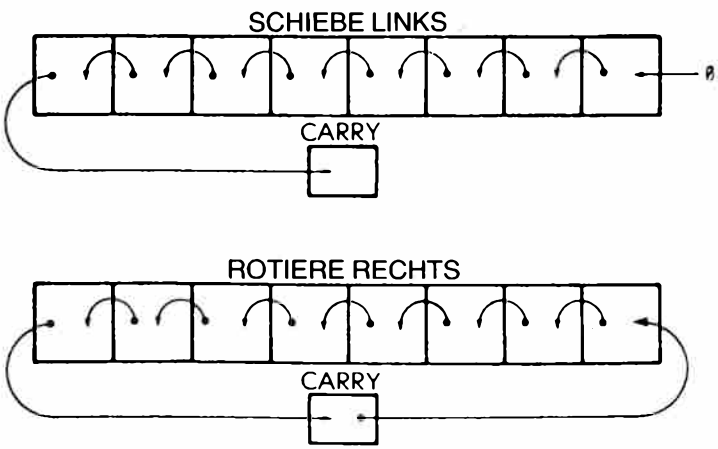


Abb. 4.1: Schieben und Rotieren

anderen Seite hereinkommt. Bei den meisten Mikroprozessoren ist dies nicht vorgesehen, es ist jedoch beim Z80 verfügbar (siehe Abb. 4.1).

Wenn man ein Wort nach rechts schieben will, ist es schließlich von Vorteil, wenn es eine weitere Art von Schieben gibt, die sich Vorzeichenstreckung oder „Arithmetisches-Rechts-Schieben" nennt. Bearbeitet man Zweierkomplemente, speziell bei der Durchführung von Gleitkommaroutinen, muß man oft negative Zahlen nach rechts schieben. Schiebt man ein Zweierkomplement nach rechts, sollte auf der linken Seite eine „1" hereinkommen (das Vorzeichen sollte beim mehrfachen Schieben so oft wie nötig reproduziert werden). Dies ist ein „Arithmetisches-Rechts-Schieben".

Tests und Sprünge

Der Testbefehl testet Bits in dem spezifizierten Register auf „0", auf „1" oder auf Kombinationen. Es muß mindestens möglich sein, das Flag-Register zu testen. Dazu ist es wünschenswert, möglichst viele Flags in diesem Register zu haben. Zusätzlich ist es vorteilhaft, wenn man mit einem Befehl Kombinationen solcher Bits testen kann. Schließlich ist es nützlich, wenn man *jedes Bit in jedem Register* testen kann, und wenn man die Inhalte zweier beliebiger Register vergleichen kann (größer, kleiner oder gleich). Üblicherweise beschränken sich die Testbefehle bei Mikroprozessoren auf das Testen einzelner Bits im Flag-Register. Der Z80 jedoch bietet weitergehende Möglichkeiten als die meisten anderen Mikroprozessoren.

Die Sprungbefehle, die üblicherweise verfügbar sind, zerfallen in drei Gruppen:

1 – den Sprung auf eine vollständige 16-Bit-Adresse
2 – den relativen Sprung, der oft auf eine 8-Bit-Distanz beschränkt ist
3 – den Aufruf von Unterprogrammen

Nützlich ist es, wenn es Sprünge nach zwei oder gar drei Zielen gibt, z. B. abhängig davon, ob das Ergebnis eines Vergleichs „größer", „kleiner" oder „gleich" ist. Günstig ist auch, wenn es kleine Sprünge gibt, die um nur wenige Befehle vorwärts oder rückwärts springen. Allerdings ist ein „kleiner Sprung" einem „Sprung" äquivalent. Bei den meisten Schleifen schließlich ist am Ende ein Dekrementier- oder ein Inkrementierbefehl, gefolgt von einem Test und einem Sprung. Wenn es einen einzelnen Befehl gibt, der inkrementiert oder dekrementiert, vergleicht und springt, dann ist das ein wesentlicher Vorteil, wenn man leistungsfähige Schleifen programmieren will. Auf den meisten Mikroprozessoren gibt es so etwas nicht. Nur einfache Verzweigungen kombiniert mit einfachen Tests sind möglich. Das erschwert natürlich die Programmierung und verringert die Leistungsfähigkeit. Beim Z80 gibt es einen Befehl „dekrementiere und springe", der allerdings nur ein spezielles Register (B) auf Null testet.

Eingabe und Ausgabe

Eingabe- und Ausgabebefehle sind spezielle Anweisungen zum Bedienen von Ein-/Ausgabegeräten. Tatsächlich betreibt die Mehrheit der 8-Bit-Mikroprozessoren *„memory mapped I/O"*: die Ein-/Ausgabegeräte werden wie Speicherbausteine an den Adreßbus angeschlossen und adressiert. Dem Programmierer erscheinen sie wie Speicherplätze. Normalerweise benötigen alle Operationen mit dem Speicher 3 Byte und sind deshalb langsam. Unter diesen Umständen ist es für eine wirksame Behandlung der Ein-und Ausgaben günstig, wenn es spezielle Techniken der kurzen Adressierung gibt, so daß Ein-/Ausgabegeräte, deren Bearbeitungszeiten kurz sein sollen, in Seite 0 liegen können. Wenn es eine spezielle Adressierung für die Seite 0 gibt, dann verwendet man diese jedoch normalerweise für RAM-Speicher, was der Verwendung für Ein-/Ausgabegeräte entgegensteht. Wie der 8080 ist auch der Z80 mit speziellen Ein-/Ausgabebefehlen ausgestattet. So hat der Entwickler im Falle des Z80 die Auswahl zwischen beiden Methoden: Ein-/Ausgabegeräte können wie Speicher adressiert werden oder aber als Ein-/Ausgabegeräte mit den speziellen Ein-/Ausgabebefehlen. Ein-/Ausgabebefehle werden an späterer Stelle in diesem Kapitel beschrieben.

Steuerbefehle

Steuerbefehle liefern Signale zur Synchronisation und können ein Programm anhalten oder unterbrechen. Sie können wie ein Zwischenstopp (englisch: break) oder als simulierter Interrupt funktionieren. (Interrupts werden in Kapitel 6 unter Ein-/Ausgabetechniken beschrieben.)

Der Befehlssatz des Z80

Einführung

Der Mikroprozessor Z80 wurde entwickelt, um den 8080 zu ersetzen und um zusätzliche Möglichkeiten zu bieten. Als Ergebnis dieser Philosophie besitzt der Z80 alle Befehle des 8080 sowie zusätzliche Befehle. Mit Blick auf die begrenzte Zahl von Bits, die in einem 8-Bit-Opcode zur Verfügung stehen, mag man sich darüber wundern, wie es die Entwickler des Z80 schafften, viele zusätzliche Befehle einzubauen. Sie taten dies, indem sie einige wenige unbelegte 8080-Opcodes verwendeten, und indem sie für die indizierten Befehle den Opcode um ein zusätzliches Byte erweiterten. Deshalb belegen manche Z80-Befehle im Speicher bis zu vier Byte.

Es ist wichtig, sich daran zu erinnern, daß man ein Programm auf viele verschiedene Arten schreiben kann. Wenn man effiziente Programmierung zustande bringen will, dann ist es unerläßlich, den Befehlssatz vollständig zu kennen und zu verstehen. Wenn man jedoch die Programmierung erlernen will, dann ist es nicht wesentlich, optimierte Programme zu schreiben. Wenn man dieses Kapitel zum ersten Mal liest, ist es unnötig, sich alle verschiedenen Befehle sofort zu merken.Es ist wichtig, daß

man die verschiedenen Arten von Befehlen einfach kennenlernt und typische Beispiele studiert. Wenn der Leser dann Programme schreibt, sollte er die Beschreibung des Z80-Befehlssatzes zu Rate ziehen und die Befehle auswählen, die sich für sein Problem am besten eignen. Deshalb wird in diesem Abschnitt ein Überblick über die verschiedenen Befehle gegeben, mit der Absicht, sie vereinfacht darzustellen und in logische Einheiten zu gruppieren. Der Leser, der die Möglichkeiten der verschiedenen Befehle erfahren will, sei auf die Einzelbeschreibungen der Befehle verwiesen.

Wir werden jetzt die Möglichkeiten, die der Z80 bietet, aus der Sicht der fünf Befehlsklassen untersuchen, die am Anfang dieses Kapitels definiert wurden.

Befehle zum Transfer von Daten

Befehle zum Datentransfer kann man auf dem Z80 in vier Gruppen einteilen: 8-Bit-Transfer, 16-Bit-Transfer, Stapelbefehle und Blocktransfer. Wir wollen sie untersuchen.

Acht-Bit-Transfer

Jeder Transfer von 8-Bit-Daten wird mit einem Ladebefehl ausgeführt. Ladebefehle haben folgendes Format:

```
LD  Ziel,Quelle
```

Beispielsweise kann man den Akkumulator mit folgendem Befehl aus dem Register B laden:

```
LD  A,B
```

Direkte Übertragungen können zwischen beliebigen Arbeitsregistern (ABCDEHL) vorgenommen werden.

Um irgendeines der Arbeitsregister, mit Ausnahme des Akkumulators, aus einer Speicherstelle zu laden, muß die Speicheradresse zuerst in ein beliebiges Registerpaar geladen werden, z. B. nach HL.

Um beispielsweise das Register C aus der Speicherstelle 1234 zu laden, muß man zuerst die Register H und L mit dem Wert „1234" laden. (Dazu wird ein 16-Bit-Ladebefehl verwendet. Dies wird im nächsten Abschnitt beschrieben.)

Dann wird der Befehl LD C,(HL) verwendet und liefert das gewünschte Ergebnis.

Der Akkumulator bildet eine Ausnahme. Er kann aus jeder ausgewählten Speicherstelle direkt geladen werden. Dies nennt man erweiterte Adressierung. Um beispielsweise den Akkumulator aus der Speicherzelle 1234 zu laden, verwendet man folgenden Befehl:

```
LD A,(1234H)   (Beachten Sie: Die Verwendung von „()"
bedeutet „Inhalt von".)
```

Dieser Befehl wird folgendermaßen im Speicher abgelegt:

Adresse PC : 3A (Opcode)
PC + 1: 34 (untere Hälfte der Adresse)
PC + 2: 12 (obere Hälfte der Adresse)

Beachten Sie, daß die Adresse innerhalb des Befehls in „umgekehrter Reihenfolge" gespeichert wird:

3A	untere Adr.	obere Adr.

Alle Arbeitsregister können auch mit einem beliebigen festgelegten 8-Bit-Wert oder „Literal" geladen werden, der im zweiten Byte des Befehls steht (dies nennt man *unmittelbare Adressierung*). Ein Beispiel ist:

LD E,12H

das den hexadezimalen Wert 12 ins Register E lädt.

Im Speicher steht der Befehl als:

PC : 1E (Opcode)
PC + 1: 12 (Unmittelbarer Operand: Literal)

Als Ergebnis dieses Befehls steht der unmittelbare Operand in dem Register E.

QUELLE

ZIEL		IMPLIED		REGISTER							REG INDIRECT			INDEXED		EXT ADDR	IMME.
		I	R	A	B	C	D	E	H	L	(HL)	(BC)	(DE)	(IX + d)	(IY + d)	(nn)	n
REGISTER	A	ED 57	ED 5F	7F	78	79	7A	7B	7C	7D	7E	0A	1A	DD 7E d	FD 7E d	3A n n	3E n
	B			47	40	41	42	43	44	45	46			DD 46 d	FD 46 d		06 n
	C			4F	48	49	4A	4B	4C	4D	4E			DD 4E d	FD 4E d		0E n
	D			57	50	51	52	53	54	55	56			DD 56 d	FD 56 d		16 n
	E			5F	58	59	5A	5B	5C	5D	5E			DD 5E d	FD 5E d		1E n
	H			67	60	61	62	63	64	65	66			DD 66 d	FD 66 d		26 n
	L			6F	68	69	6A	6B	6C	6D	6E			DD 6E d	FD 6E d		2E n
REG INDIRECT	(HL)			77	70	71	72	73	74	75							36 n
	(BC)			02													
	(DE)			12													
INDEXED	(IX+d)			DD 77 d	DD 70 d	DD 71 d	DD 72 d	DD 73 d	DD 74 d	DD 75 d							DD 36 d n
	(IY+d)			FD 77 d	FD 70 d	FD 71 d	FD 72 d	FD 73 d	FD 74 d	FD 75 d							FD 36 d n
EXT ADDR	(nn)			32 n n													
IMPLIED	I			ED 47													
	R			ED 4F													

Abb. 4.2: Acht-Bit-Ladebefehle – LD

Zum Laden von Registerinhalten steht auch die *indizierte Adressierung* zur Verfügung, und sie wird im nächsten Kapitel über Adressierungstechniken vollständig beschrieben. Zum Laden spezieller Register gibt es verschiedene andere Möglichkeiten. Die Tabelle Abb. 4.2 zeigt alle Möglichkeiten (die Tabelle wurde von Zilog veröffentlicht). Die Befehle, die auch auf dem 8080 verfügbar sind, sind grau unterlegt.

16-Bit-Transfer

Grundsätzlich kann man jedes der 16-Bit-Registerpaare BC, DE, HL, SP, IX und IY mit einem direkten 16-Bit-Operanden, von einer festgelegten Speicheradresse (*erweiterte Adressierung*) oder von dem obersten Stapelelement, d. h. von der Adresse in SP geladen werden. Umgekehrt kann man den Inhalt dieser Registerpaare auf die gleiche Art und Weise bei einer festgelegten Speicheradresse oder auf dem Stapel ablegen. Zusätzlich kann man das Register SP aus HL, IX oder IY laden. Dies erleichtert den Umgang mit mehreren Stapeln. Das Regiterpaar AF kann man auch auf dem Stapel ablegen.

Die Tabelle in Abb. 4.3 listet alle Möglichkeiten auf. Die Stapeloperationen Push und Pop sind als 16-Bit-Datentransfers mit eingeschlossen.

ZIEL REGISTER	QUELLE REGISTER AF	BC	DE	HL	SP	IX	IY	IMM. EXT. nn	EXT. ADDR. (nn)	REG. INDIR. (SP)
AF										F1
BC								01 n n	ED 4B n n	C1
DE								11 n n	ED 5B n n	D1
HL								21 n n	2A n n	E1
SP				F9		DD F9	FD F9	31 n n	ED 7B n n	
IX								DD 21 n n	DD 2A n n	DD E1
IY								FD 21 n n	FD 2A n n	FD E1
EXT. ADDR. (nn)		ED 43 n n	ED 53 n n	22 n n	ED 73 n n	DD 22 n n	FD 22 n n			
REG. IND. (SP)	F5	C5	D5	E5		DD E5	FD E5			

PUSH BEFEHLE →

ACHTUNG: Die Befehle PUSH und POP passen den SP nach jedem Befehl entsprechend an.

POP BEFEHLE

Abb. 4.3: 16-Bit-Ladebefehle – 'LD, 'PUSH' und 'POP'

Alle Stapeloperationen übertragen den Inhalt eines Registerpaares zum oder vom Stapel. Beachten Sie, daß es keine Push- und Pop-Befehle gibt, mit denen man Acht-Bit-Register einzeln ablegen kann.

Ein Push oder Pop von zwei Bytes wird immer mit einem Registerpaar durchgeführt: mit AF, BC, DE, HL, IX oder IY (siehe die untere Zeile und die rechte Spalte der Abbildung 4.3).

Wenn man mit den Registerpaaren AF, BC, DE oder HL arbeitet, besteht der Befehl aus einem Byte, wodurch eine gute Effizienz erreicht wird. Als Beispiel wollen wir annehmen, daß der Stapelzeiger SP den Wert „0100" enthält. Dann wird der folgende Befehl ausgeführt:

```
PUSH  AF
```

Wenn der Inhalt dieses Registerpaares auf den Stapel abgelegt wird, wird zuerst der Stapelzeiger dekrementiert und dann der Inhalt des Registers A auf dem Stapel abgelegt. Dann wird SP wieder dekrementiert und der Inhalt von F auf dem Stapel abgelegt. Am Ende der Übertragung zum Stapel zeigt SP auf das oberste Element des Stapels, in unserem Beispiel steht dort der Inhalt von F.

Es ist wichtig, sich daran zu erinnern, daß der SP beim Z80 auf das *oberste Stapelelement* zeigt, und daß der SP immer *dekremiert* wird, wenn ein Registerpaar abgelegt wird. Oft werden andere Vereinbarungen verwendet, und das kann zu Unklarheiten führen.

Austauschbefehle

Zusätzlich wurde eine spezielle Abkürzung EX für Austauschoperationen reserviert. EX ist keine einfache Datenübertragung. Es vertauscht tatsächlich den Inhalt zweier festgelegter Plätze. EX kann z. B. verwen-

		IMPLIZIERTE ADRESSIERUNG				
		AF'	BC', DE' & HL'	HL	IX	IY
IMPLI-ZIERT	AF	08				
	BC, DE & HL		D9			
	DE			EB		
REG. INDIR.	(SP)			E3	DD E3	FD E3

Abb. 4.4: Austausch 'EX' und 'EXX'

det werden, um das oberste Stapelelement mit HL, IX oder IY zu vertauschen oder auch den Inhalt von DE und HL bzw. von AF und AF' (AF' bedeutet das zweite AF-Registerpaar im Z80).

Schließlich gibt es einen speziellen Befehl EXX, der den Inhalt von BC, DE und HL mit dem Inhalt der entsprechenden Zweitregister im Z80 vertauscht.

Die möglichen Vertauschungen zeigt Abb. 4.4

			QUELLE: REG. INDIR.	QUELLE: (HL)
ZIEL	REG. INDIR.	(DE)	ED A0	'LDI' – Load (DE)◄—(HL) Inkr HL & DE, Dekr BC
			ED B0	'LDIR,' – Load (DE)◄—(HL) Inkr HL & DE, Dekr BC, wiederhole, bis BC = 0
			ED A8	'LDD' – Load (DE)◄—(HL) Dec HL & DE, Dec BC
			ED B8	'LDDR' – Load (DE)◄—(HL) Dekr HL & DE, Dekr BC, wiederhole, bis BC = 0

Reg HL zeigt auf die Quelle
Reg DE zeigt auf das Ziel
Reg BC ist der Bytezähler

Abb. 4.5: Blocktransferbefehle

Blocktransferbefehle

Blocktransferbefehle sind Befehle, die einen ganzen Block von Daten übertragen und nicht nur ein oder zwei Bytes. Für den Hersteller sind Blocktransferbefehle schwieriger einzubauen, als die meisten anderen Befehle und sie sind üblicherweise in Mikroprozessoren nicht verfügbar. Sie sind bei der Programmierung nützlich und können die Leistungsfähigkeit eines Programms erhöhen, speziell bei Ein-/Ausgabebefehlen. Ihre Verwendung und ihre Vorteile werden überall in diesem Buch demonstriert. Beim Z80 sind einige spezielle Blocktransferbefehle verfügbar. Sie verwenden spezielle Vereinbarungen.

Alle Blocktransferbefehle verlangen die Verwendung von drei Registerpaaren: BC, DE und HL.

BC wird als 16-Bit-Zähler verwendet. Dies heißt, daß bis zu $2^{16} = 64k$ Byte automatisch übertragen werden können. HL wird als Zeiger auf die Quelle benutzt. Es kann auf eine beliebige Stelle im Speicher zeigen. DE

dient als Zeiger auf das Ziel und kann ebenfalls auf eine beliebige Stelle im Speicher zeigen.

Es gibt vier Blocktransferbefehle:

LDD, LDDR, LDI und LDIR.

Jeder dieser Befehle dekrementiert den Zähler BC bei jedem Transfer. Zwei von ihnen, LDD und LDDR, dekrementieren die Zeiger DE und HL, während die beiden anderen Befehle (LDI und LDIR) DE und HL inkrementieren. Bei jeder der beiden Gruppen von Befehlen bedeutet der Buchstabe R am Ende der Abkürzung eine automatische Wiederholung. Wir wollen diese Befehle näher untersuchen.

LDI bedeutet „Lade und inkrementiere". Er überträgt ein Byte von der Speicherstelle, auf die H und L zeigt, zu dem Ziel im Speicher, auf das D und E zeigen. Er dekrementiert dann BC. HL und DE werden automatisch inkrementiert, so daß alle Register für eine eventuelle weitere Übertragung richtig vorbereitet sind.

LDIR heißt: „Lade, inkrementiere und wiederhole", d. h. führe LDI wiederholt aus, bis das Zählerregister BC den Wert „0" erreicht. Er wird verwendet, um einen fortlaufenden Block von Daten aus einem Speicherbereich automatisch in einen anderen zu übertragen.

LDD und LDDR arbeiten auf die gleiche Weise, außer daß die Adreßzeiger dekrementiert und nicht inkrementiert werden. Der Transfer beginnt deshalb bei der höchsten Adresse des Blocks und nicht bei der niedrigsten. Die Wirkungsweise der vier Befehle ist in Abb. 4.5 zusammengefaßt.

Ähnliche automatische Befehle gibt es auch für CP (Vergleiche). Sie sind in Abb. 4.6 zusammengefaßt.

SUCHADRESSE

REG. INDIR.	
(HL)	
ED A1	'CPI' Inkr HL, Dekr BC
ED B1	'CPIR', Inkr HL, Dekr BC wiederhole, bis BC = 0 oder Gleichheit gefunden
ED A9	'CPD' Dekr HL & BC
ED B9	'CPDR' Dekr HL & BC wiederhole, bis BC = 0 oder Gleichheit gefunden

HL zeigt auf die Speicherstelle, deren Inhalt mit dem Akkumulator verglichen werden soll.
BC ist der Bytezähler

Abb. 4.6: Blocksuchbefehle

Befehle zur Datenverarbeitung

Arithmetik

Zwei hauptsächliche Arithmetikbefehle stehen zur Verfügung: Addition und Subtraktion. Sie wurden im vorhergehenden Kapitel ausgiebig angewendet. Es gibt zwei Typen der Additon, mit und ohne Übertrag: ADC und ADD. Ebenso stehen zwei Subtraktionsbefehle zur Verfügung, mit und ohne Übertrag. Diese sind SBC und SUB.

Zusätzlich gibt es drei Spezialbefehle: DAA, CPL und NEG. Der Befehl Dezimalanpassung (DAA) wurde dazu verwendet, die BCD-Operationen einzubauen. Er wird normalerweise bei jeder BCD-Addition oder -Subtraktion benutzt. Außerdem gibt es zwei Befehle zur Komplementbildung. CPL bildet das Einerkomplement des Akkumulators und NEG wandelt den Akkumulator in sein Zweierkomplement um.

Alle bisherigen Befehle rechnen mit Acht-Bit-Daten. 16-Bit-Befehle sind eingeschränkter. ADD, ADC und SBC sind für bestimmte Register verfügbar, wie Abb. 4.8 zeigt.

QUELLE

	REGISTER ADDRESSING							REG. INDIR.	INDEXED		IMMED.
	A	B	C	D	E	H	L	(HL)	(IX+d)	(IY+d)	n
'ADD'	87	80	81	82	83	84	85	86	DD 86 d	FD 86 d	C6 n
ADD w CARRY 'ADC'	8F	88	89	8A	8B	8C	8D	8E	DD 8E d	FD 8E d	CE n
SUBTRACT 'SUB'	97	90	91	92	93	94	95	96	DD 96 d	FD 96 d	D6 n
SUB w CARRY 'SBC'	9F	98	99	9A	9B	9C	9D	9E	DD 9E d	FD 9E d	DE n
'AND'	A7	A0	A1	A2	A3	A4	A5	A6	DD A6 d	FD A6 d	E6 n
'XOR'	AF	A8	A9	AA	AB	AC	AD	AE	DD AE d	FD AE d	EE n
'OR'	B7	B0	B1	B2	B3	B4	B5	B6	DD B6 d	FD B6 d	F6 n
COMPARE 'CP'	BF	B8	B9	BA	BB	BC	BD	BE	DD BE d	FD BE d	FE n
INCREMENT 'INC'	3C	04	0C	14	1C	24	2C	34	DD 34 d	FD 34 d	
DECREMENT 'DEC'	3D	05	0D	15	1D	25	2D	35	DD 35 d	FD 35 d	

Abb. 4.7: Acht-Bit Arithmetik und Logik

Schließlich gibt es für alle Register Befehle zum Inkrementieren und Dekrementieren, sowohl für das Acht-Bit- als auch für das 16-Bit-Format. Sie sind in Abb. 4.7 (Acht-Bit-Befehle) und in Abb. 4.8 (16-Bit-Befehle) angegeben.

Beachten Sie, daß im allgemeinen alle arithemtischen Operationen einige Flags beeinflussen. Ihre Wirkungsweise ist bei der Beschreibung der Befehle an späterer Stelle in diesem Kapitel im einzelnen erklärt. Es ist jedoch wichtig zu wissen, daß die Befehle INC und DEC keine Flags beeinflussen, wenn sie auf Registerpaare angewendet werden. Es ist wichtig, sich diese Einzelheit zu merken. Das bedeutet, daß das Z-Bit im Flagregister nicht gesetzt wird, wenn eines der Registerpaare auf den Wert „0" dekrementiert oder inkrementiert wird. Der Inhalt des Registerpaares muß in dem Programm ausdrücklich auf den Wert Null getestet werden.

Es ist auch wichtig zu wissen, daß die Befehle ADC und SBC immer auf alle Flags wirken. Dies heißt nicht, daß nach der Ausführung unbedingt alle Flags geändert sein müssen. Sie können aber verändert sein.

Logik

Drei logische Befehle sind vorgesehen: AND, OR (inklusiv) und XOR (exklusiv), sowie ein Vergleichsbefehl CP. Sie arbeiten alle nur mit 8-

		QUELLE					
		BC	DE	HL	SP	IX	IY
ZIEL 'ADD'	HL	09	19	29	39		
	IX	DD 09	DD 19		DD 39	DD 29	
	IY	FD 09	FD 19		FD 39		FD 29
ADD WITH CARRY AND SET FLAGS 'ADC'	HL	ED 4A	ED 5A	ED 6A	ED 7A		
SUB WITH CARRY AND SET FLAGS 'SBC'	HL	ED 42	ED 52	ED 62	ED 72		
INCREMENT 'INC'		03	13	23	33	DD 23	FD 23
DECREMENT 'DEC'		0B	1B	2B	3B	DD 2B	FD 2B

Abb. 4.8: 16-Bit Arithmetik und Logik

Bit-Daten. Wir wollen sie der Reihe nach untersuchen. (Eine Tabelle, die alle Möglichkeiten und die Opcodes für diese Befehle angibt, ist in Abb. 4.7 enthalten.)

AND

Jede logische Operation wird durch eine *Wahrheitstabelle* charakterisiert, die den logischen Wert des Ergebnisses als Funktion der Eingaben ausdrückt. Die Wahrheitstabelle für AND erscheint unten:

0 AND 0 = 0
0 AND 1 = 0
1 AND 0 = 0
1 AND 1 = 1

oder

AND	0	1
0	0	0
1	0	1

Die Operation „AND" wird dadurch festgelegt, daß der Ausgang nur dann „1" ist, wenn beide Eingänge „1" sind. Mit anderen Worten, wenn einer der Eingänge „0" ist, dann ist garantiert, daß das Ergebnis „0" ist. Diese Eigenschaft wendet man an, um ein Bit in einem Wort auf Null zu setzen. Man nennt dies „Maskieren".

Eine der wesentlichen Anwendungen des Befehls AND ist es, eine oder mehrere festgelegte Stellen in einem Wort zu löschen oder auszublenden. Wir wollen beispielsweise annehmen, daß wir die vier rechten Bits eines Worts zu Null setzen wollen. Dies erledigt das folgende Programm:

```
LD      A,WORT    WORT enthält „10101010"
AND     11110000B „11110000" ist die Maske
```

Wir wollen annehmen, daß WORT gleich „10101010" ist. Als Ergebnis dieses Programms steht „10100000" im Akkumulator. „B" kennzeichnet eine Dualzahl.

Aufgabe 4.1: Schreibe ein dreizeiliges Programm, das die Bits 1 und 6 von WORT auf Null setzt.

Aufgabe 4.2: Was passiert mit der Maske „11111111"?

OR

Dieser Befehl ist das inklusive OR. Er wird durch folgende Wahrheitstabelle charakterisiert:

0 OR 0 = 0
0 OR 1 = 1
1 OR 0 = 1
1 OR 1 = 1

oder

OR	0	1
0	0	1
1	1	1

Das logische OR wird dadurch charakterisiert, daß das Ergebnis immer „1" ist, wenn irgendeiner der Eingänge „1" ist. Eine Anwendung, die auf der Hand liegt, ist es, Bits in einem Wort auf „1" zu setzen.

Wir wollen die rechten vier Bits von WORT zu Einsen machen. Das Programm dafür ist:

```
LD   A,WORT
OR   00001111B
```

Wir wollen annehmen, daß WORT „10101010" enthielt. Im Akkumulator steht dann das Ergebnis „10101111".

Aufgabe 4.3: Was würde passieren, wenn wir den Befehl OR 10101010B verwenden?

Aufgabe 4.4: Was bewirkt ein OR mit hexadezimal „FF"?

XOR

XOR bedeutet „exklusives OR". Das exklusive OR unterscheidet sich vom inklusiven OR, das wir gerade beschrieben haben, in einem Punkt: Das Ergebnis ist „1", wenn einer und nur einer der Operanden „1" ist. Sind beide Operanden „1", dann ergibt das normale OR das Ergebnis „1". Das exklusive OR ergibt eine „0". Die Wahrheitstabelle ist:

0 XOR 0 = 0
0 XOR 1 = 1
1 XOR 0 = 1
1 XOR 1 = 0

oder

XOR	0	1
0	0	1
1	1	0

Das exklusive OR verwendet man für Vergleiche. Das XOR zweier Worte ist nicht Null, wenn irgendein Bit verschieden ist.

Außerdem kann man mit dem exklusiven OR den Akkumulator oder einzelne Stellen im Akkumulator komplementieren. Das Programm

```
LD    A,WORT
XOR   11111111B
```

liefert im Akkumulator das Komplement von WORT.

Aufgabe 4.5: Welches Ergebnis liefert XOR, wenn WORT hexadezimal „00" enthält?

Schieben und Rotieren

Zuerst wollen wir zwischen den Operationen Schieben und Rotieren unterscheiden, die in Abb. 4.9 veranschaulicht sind. Beim Schieben wird der Inhalt des Registers um eine Stelle nach links oder nach rechts geschoben. Das Bit, das herausfällt, kommt ins Übertragsbit und das Bit, das hereinkommt, ist Null. Dies wurde im vorhergehenden Kapitel erklärt.

Es gibt eine Ausnahme: Das *Arithmetische-Rechts-Schieben.* Wenn man mit negativen Zahlen in der Darstellung als Zweierkomplement rechnet, dann ist das linke Bit das Vorzeichenbit. Bei negativen Zahlen ist

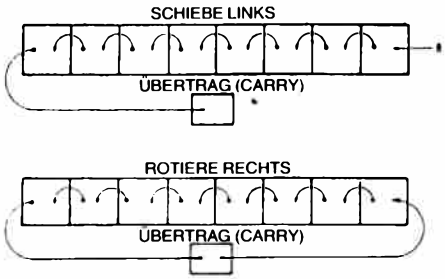


Abb. 4.9: Schieben und Rotieren

dies „1". Dividiert man eine negative Zahl durch „2", indem man sie nach rechts schiebt, so sollte das Ergebnis negativ bleiben, d. h. das linke Bit sollte „1" bleiben. Dies macht der Befehl SRA oder „Schiebe Rechts Arithmetisch". Bei diesem arithmetischen Rechts-Schieben ist das Bit, das links hereinkommt, mit dem Vorzeichenbit identisch. Es ist „0", wenn das linke Bit eine „0" war, und es ist „1", wenn das linke Bit eine „1" war. Dies wird in der Abb. 4.10 rechts dargestellt, die die möglichen Schiebe- und Rotieroperationen zeigt.

Rotationen

Eine Rotation unterscheidet sich von einem Schieben dadurch, daß das hereinkommende Bit entweder das ist, das auf der anderen Seite herausfällt, oder das Übertragsbit. Beim Z80 gibt es zwei Arten von Rotationen: eine Acht-Bit-Rotation und eine Neun-Bit-Rotation.

Das Neun-Bit-Rotieren ist in Abb.4.11 dargestellt. Bei einer Rotation nach rechts werden die acht Bits des Registers beispielsweise um eine

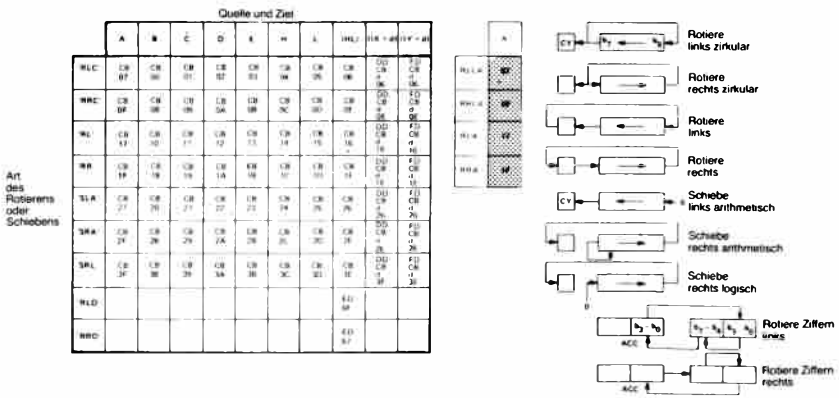


Abb. 4.10: Rotieren und Schieben

Stelle nach rechts verschoben. Das Bit, das rechts herausfällt, kommt wie üblich ins Übertragsbit. Das Bit, das jetzt links hereinkommt ist das alte Übertragungsbit (bevor es von dem herausfallenden Bit überschrieben wurde). Mathematisch nennt man das eine Neun-Bit-Rotation, da die acht Bit des Registers und das neunte Bit (das Übertragsbit) um eine Stelle nach rechts rotiert wurden. Eine Rotation nach links liefert entsprechend das gleiche Ergebnis in umgekehrter Richtung.

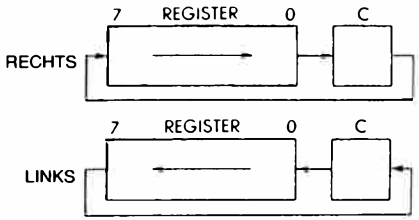


Abb. 4.11: Neun-Bit-Rotation

Die Acht-Bit-Rotation funktioniert ähnlich. Abhängig von der Rotationsrichtung wird Bit 0 nach Bit 7 oder Bit 7 nach Bit 0 kopiert. Zusätzlich wird das herausfallende Bit auch ins Übertragsbit kopiert. In Abbildung 4.12 ist dies veranschaulicht.

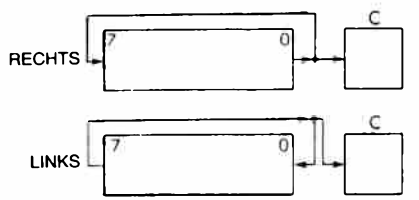


Abb. 4.12: Acht-Bit-Rotation

Spezielle Befehle für Ziffern

Um das Rechnen mit BCD-Zahlen zu erleichtern, sind zwei spezielle Rotierbefehle für (BCD-) Ziffern vorgesehen. Das Ergebnis ist eine Rotation um vier Bit zwischen zwei Ziffern in der Speicherstelle, auf die die Register HL zeigen, und einer Ziffer in der unteren Hälfte des Akkumulators. Abbildung 4.13 veranschaulicht das.

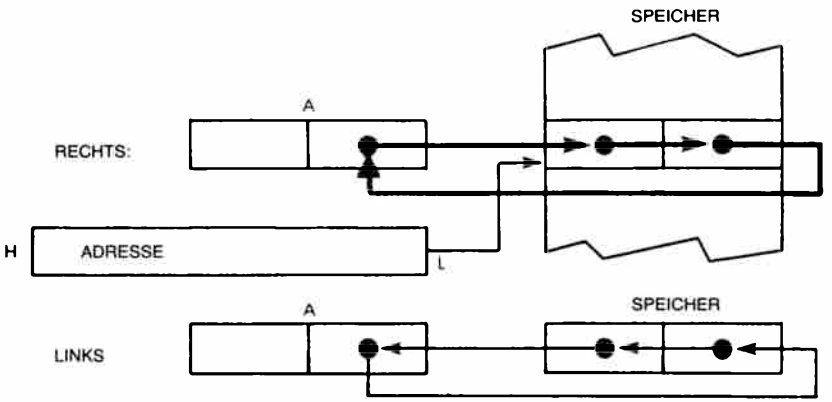


Abb. 4.13: Rotieren von Ziffern

Bit-Verarbeitung

Oben wurde gezeigt, wie man die logischen Operationen dazu verwenden kann, um einzelne Bits oder Gruppen von Bits im Akkumulator zu setzen oder zurückzusetzen. Es ist jedoch nützlich, wenn man mit einem Befehl ein beliebiges Bit in einem beliebigen Register oder einer Speicherzelle setzen oder zurücksetzen kann. Diese Möglichkeit beansprucht eine erhebliche Zahl von Opcodes und steht deshalb bei den meisten Mikroprozessoren nicht zur Verfügung. Der Z80 ist jedoch mit ausgiebigen Möglichkeiten zur Bitmanipulation ausgerüstet. Diese werden in Abb. 4.14 gezeigt. Diese Tabelle enthält auch die Testbefehle, die im nächsten Abschnitt beschrieben werden.

Zur Operation mit dem Übertragsflag gibt es zwei spezielle Befehle: CCF (Komplementiere das Übertragsflag) und SCF (Setze das Übertragsflag). Sie sind in Abb. 4.15 gezeigt.

Tests und Sprünge

Da sich Testbefehle hauptsächlich auf das Flagregister beziehen, werden wir hier die Rolle jedes einzelnen Flags genau beschreiben. Der Inhalt des Flagregisters erscheint in Abb. 4.16.

C ist der Übertrag, N ist Addition oder Subtraktion, P/V ist Parität oder Überlauf, H ist der Halbübertrag, Z ist Null und S ist das Vorzeichen. Bit 3 und 5 des Flagregisters werden nicht benutzt („0"). Die beiden Flags H und N werden für die BCD-Arithmetik verwendet und können nicht getestet werden. Die anderen vier Flags (C, P/V, Z, S) können bei einem bedingten Sprung oder Unterprogrammaufruf getestet werden.

		ADRESSIERTES REGISTER							REG. INDIR.	INDEXED	
	BIT	A	B	C	D	E	H	L	(HL)	(IX+d)	(IY+d)
TEST 'BIT'	0	CB 47	CB 40	CB 41	CB 42	CB 43	CB 44	CB 45	CB 46	DD CB d 46	FD CB d 46
	1	CB 4F	CB 48	CB 49	CB 4A	CB 4B	CB 4C	CB 4D	CB 4E	DD CB d 4E	FD CB d 4E
	2	CB 57	CB 50	CB 51	CB 52	CB 53	CB 54	CB 55	CB 56	DD CB d 56	FD CB d 56
	3	CB 5F	CB 58	CB 59	CB 5A	CB 5B	CB 5C	CB 5D	CB 5E	DD CB d 5E	FD CB d 5E
	4	CB 67	CB 60	CB 61	CB 62	CB 63	CB 64	CB 65	CB 66	DD CB d 66	FD CB d 66
	5	CB 6F	CB 68	CB 69	CB 6A	CB 6B	CB 6C	CB 6D	CB 6E	DD CB d 6E	FD CB d 6E
	6	CB 77	CB 70	CB 71	CB 72	CB 73	CB 74	CB 75	CB 76	DD CB d 76	FD CB d 76
	7	CB 7F	CB 78	CB 79	CB 7A	CB 7B	CB 7C	CB 7D	CB 7E	DD CB d 7E	FD CB d 7E
RESET BIT 'RES'	0	CB 87	CB 80	CB 81	CB 82	CB 83	CB 84	CB 85	CB 86	DD CB d 86	FD CB d 86
	1	CB 8F	CB 88	CB 89	CB 8A	CB 8B	CB 8C	CB 8D	CB 8E	DD CB d 8E	FD CB d 8E
	2	CB 97	CB 90	CB 91	CB 92	CB 93	CB 94	CB 95	CB 96	DD CB d 96	FD CB d 96
	3	CB 9F	CB 98	CB 99	CB 9A	CB 9B	CB 9C	CB 9D	CB 9E	DD CB d 9E	FD CB d 9E
	4	CB A7	CB A0	CB A1	CB A2	CB A3	CB A4	CB A5	CB A6	DD CB d A6	FD CB d A6
	5	CB AF	CB A8	CB A9	CB AA	CB AB	CB AC	CB AD	CB AE	DD CB d AE	FD CB d AE
	6	CB B7	CB B0	CB B1	CB B2	CB B3	CB B4	CB B5	CB B6	DD CB d B6	FD CB d B6
	7	CB BF	CB B8	CB B9	CB BA	CB BB	CB BC	CB BD	CB BE	DD CB d BE	FD CB d BE
SET BIT 'SET'	0	CB C7	CB C0	CB C1	CB C2	CB C3	CB C4	CB C5	CB C6	DD CB d C6	FD CB d C6
	1	CB CF	CB C8	CB C9	CB CA	CB CB	CB CC	CB CD	CB CE	DD CB d CE	FD CB d CE
	2	CB D7	CB D0	CB D1	CB D2	CB D3	CB D4	CB D5	CB D6	DD CB d D6	FD CB d D6
	3	CB DF	CB D8	CB D9	CB DA	CB DB	CB DC	CB DD	CB DE	DD CB d DE	FD CB d DE
	4	CB E7	CB E0	CB E1	CB E2	CB E3	CB E4	CB E5	CB E6	DD CB d E6	FD CB d E6
	5	CB EF	CB E8	CB E9	CB EA	CB EB	CB EC	CB ED	CB EE	DD CB d EE	FD CB d EE
	6	CB F7	CB F0	CB F1	CB F2	CB F3	CB F4	CB F5	CB F6	DD CB d F6	FD CB d F6
	7	CB FF	CB F8	CB F9	CB FA	CB FB	CB FC	CB FD	CB FE	DD CB d FE	FD CB d FE

Abb. 4.14: Bitmanipulationen

Dezimaler Abgleich, 'DAA'	27
Komplementiere Akk., 'CPL'	2F
Negiere Akk., 'NEG' (Zweierkomplement)	ED 44
Komplementiere Carry-Flag, 'CCF'	3F
Setze Carry-Flag, 'SCF'	37

Abb. 4.15: Universelle Operationen mit AF

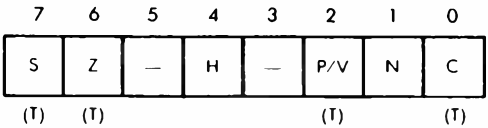


Abb. 4.16: Das Flagregister

Übertrag (Carry, C)

Bei fast allen Mikroprozessoren und speziell beim Z80 hat das Übertragsbit zwei Funktionen. Zum einen gibt es an, ob bei einer Addition oder Subtraktion ein Übertrag auftrat. Zum anderen wird es bei Schiebe- und Rotieroperationen als neuntes Bit verwendet. Die Verwendung eines einzelnen Bits für beide Aufgaben erleichtert einige Operationen, z. B. die Multiplikation. Dies sollte nach der Erklärung der Multiplikation im letzten Kapitel klar sein.

Wenn man lernt, das Übertragsbit zu verwenden, dann ist es wichtig, sich daran zu erinnern, daß es alle arithmetischen Operationen abhängig vom Ergebnis setzt oder rücksetzt. Auch alle Schiebe- und Rotierbefehle benutzen das Übertragsbit und setzen es oder setzen es zurück, abhängig von dem Bit, das aus dem Register herauskommt.

Die logischen Operationen (AND, OR, XOR) setzen das Übertragsbit immer zurück. Man kann sie dazu verwenden, um das Übertragsbit zu löschen.

Folgende Befehle beeinflussen das Übertragsbit: ADD A,s; ADC A,s; SUB s; SBC A,s; CP s; NEG; AND s; OR s; XOR s; ADD DD,ss; ADC HL,ss; SBC HL,ss; RLA; RLCA; RRA; RRCA; RL m; RLC m; RR m; RRC m; SLA m; SRA m; SRL m; DAA; SCF; CCF.

Subtraktion (N)

Dieses Flag wird normalerweise vom Programmierer nicht benutzt, sondern vom Z80 selbst bei BCD-Befehlen.Der Leser wird sich aus dem letzten Kapitel daran erinnern, daß man nach einer BCD Addition oder Subtraktion den Befehl DAA (Dezimalabgleich) ausführt, um das richtige Ergebnis zu erhalten. Der „Abgleich" funktioniert jedoch nach einer Addition anders als nach einer Subtraktion. Der Befehl DAA wirkt deshalb unterschiedlich, abhängig von dem Flag N. Nach einer Addition wird das Flag N auf „0" gesetzt, nach einer Subtraktion auf „1".

Das Zeichen, das für dieses Flag verwendet wird, „N", ist vielleicht für den Programmierer verwirrend, der mit anderen Prozessoren gearbeitet hat, weil man es mit dem Vorzeichenbit verwechseln kann. Es ist ein internes Vorzeichenbit.

N wird auf „0" gesetzt durch: ADD A,s; ADC A,s; AND s; OR s; XOR s; INC s; ADD DD,ss; ADC HL,ss; RLA; RLCA; RRA; RRCA; RL m; RLC m; RR m; RRC m; SLA m; SRA m; SRL m; RLD; RRD; SCF; CCF; IN r,(C); LDI; LDD; LDIR; LDDR; LD A,I; LD A,R; BIT b,s.

N wird auf „1" gesetzt durch SUB s; SBC A,s; CP s; NEG; DEC m; SBC HL,ss; CPL; INI; IND; OUTI; OUTD; INIR; INDR; OTIR; OTDR; CPI; CPIR; CPD; CPDR.

Parität/Überlauf (P/V)

Das Flag Parität/Überlauf erfüllt zwei verschiedene Funktionen. Bestimmte Befehle setzen dieses Flag in Abhängigkeit von der Parität des Ergebnisses. Die Parität bestimmt man, indem man die Einsen im Ergebnis zählt. Ist diese Anzahl ungerade, wird das Paritätsbit auf „0" gesetzt (ungerade Parität). Ist sie gerade, wird das Paritätsbit auf „1" gesetzt (gerade Parität). Die Parität verwendet man meistens für Blöcke von Zeichen (üblicherweise im ASCII-Format). Das Paritätsbit ist ein zusätzliches Bit, das an den 7-Bit-Kode, der das Zeichen bestimmt, angehängt wird, um festzustellen, ob die gespeicherten Daten unversehrt sind. Wurde beispielsweise ein Bit des Kodes, der das Zeichen darstellt, wegen eines Speicherfehlers (z. B. bei einem Plattenfehler oder einem Fehler im RAM-Speicher) oder bei der Übertragung verfälscht, dann hat sich die Zahl der Einsen in dem Sieben-Bit-Kode geändert. Wenn man das Paritätsbit überprüft, wird die Abweichung erkannt und ein Fehler angezeigt. Teilweise wird das Flag bei logischen und Rotieroperationen benutzt. Außerdem zeigt das Paritätsbit bei manchen Eingabeoperationen von einem Ein-/Ausgabegerät die Parität der gelesenen Daten an.

Der Leser, der sich mit dem 8080 auskennt, möge beachten, daß das Paritätsflag beim 8080 nur als solches verwendet wird. Beim Z80 wird es für verschiedene zusätzliche Funktionen benutzt. Wenn man von einem dieser Mikroprozessoren zum anderen übergeht, sollte man dieses Flag deshalb mit Vorsicht behandeln.

Beim Z80 ist die zweite wesentliche Funktion dieses Flags die eines Überlauf-Flags (das gibt es beim 8080 nicht). Das Überlauf-Flag wurde in Kapitel 1 beschrieben, als das Zweierkomplement eingeführt wurde. Es zeigt an, wenn bei einer Addition oder Subtraktion das Vorzeichenbit „fälschlicherweise" verändert wurde, da das Ergebnis in das Vorzeichenbit überlief. (Erinnern Sie sich daran, daß die größte positive Zahl +127 und die kleinste negative Zahl −128 ist, wenn man eine acht Bit lange Zweierkomplement-Darstellung verwendet.)

Letztlich wird dieses Bit beim Z80 noch für zwei andere Funktionen benutzt.

Bei den Blocktransferbefehlen (LDD, LDDR, LDI, LDIR) und den Suchbefehlen (CPD, CPDR, CPI, CPIR) wird dieses Flag verwendet, um anzuzeigen, ob das Zählerregister B den Wert „0" erreicht hat. Das Flag wird auf „0" zurückgesetzt, wenn das Register BC auf „0" dekrementiert wurde, sonst wird es auf „1" gesetzt.

Wenn man die beiden Spezialbefehle LD A,I und LD A,R ausführt, dann enthält das P/V-Flag den Wert des Interrupt-Enable-Flip-Flops (IFF2). Diese Eigenschaft kann man verwenden, um den Inhalt dieses Flip-Flops zu retten oder zu testen.

Das P-Flag wird beeinflußt von: AND s; OR s; XOR s; RL m; RLC m; RR m; RRC m; SLAm; SRA m; SRL m; RLD; RRD; DAA; IN r,(C).

Das V-Flag wird beeinflußt von: ADD A,s; ADC A,s; SUB s; SBC A,s; CP s; NEG; INC s; DEC m; ADC HL,ss; SBC HL,ss; NEG.

Außerdem wird es verwendet von: LDIR; LDDR (auf „0" gesetzt); LDI; LDD; CPI; CPIR; CPD; CPDR.

Das Halbübertrag-Flag (H)

Dieses Flag zeigt einen eventuellen Übertrag von Bit 3 nach Bit 4 bei arithmetischen Operationen an. Mit anderen Worten, es stellt den Übertrag vom unteren Nibble (Gruppe von vier Bit) in den oberen dar. Natürlich wird es vorwiegend für BCD-Operationen verwendet. Beim Befehl Dezimalabgleich (DAA) wird es innerhalb des Mikroprozessors verwendet, um das Ergebnis auf den richtigen Wert anzupassen.

Bei einer Addition wird das Flag gesetzt, wenn ein Übertrag von Bit 3 nach Bit 4 auftritt, sonst wird es zurückgesetzt. Entsprechend wird es bei einer Subtraktion gesetzt, wenn ein Bit von Bit 4 nach Bit 3 „geborgt" wird, sonst wird es zurückgesetzt.

Das Flag wird beeinflußt beim Addieren, Subtrahieren, Inkrementieren, Dekrementieren, Vergleichen und bei logischen Operationen.

Folgende Befehle beeinflußen das H-Bit: ADD A,r; ADC A,s; SUB s; SBC A,s; CP s; NEG; AND s; OR s; XOR s; INC s; DEC m; RLA; RLCA; RRA; RRCA; RL m; RLC m; RR m; RRC m; SLA m; SR m; SRL m; RLD; RRD; DAA; CPL; SCF; IN r,(C); LDI; LDD; LDIR; LDDR; LD A,I; LD A,R; BIT b,r; CPI; CPIR; CPD; CPDR.

Beachten Sie, daß das H-Bit von 16-Bit Additionen und Subtraktionen nicht beeinflußt wird.

Null (Zero, Z)

Das Flag Z verwendet man zur Anzeige, ob der Wert eines Bytes, das berechnet oder übertragen wurde, Null ist. Es wird auch bei Vergleichsoperationen verwendet, um Gleichheit anzuzeigen, und bei verschiedenen anderen Operationen.

Bei einer Operation oder einer Datenübertragung mit dem Ergebnis Null wird das Bit Z auf „1" gesetzt. Sonst wird Z auf „0" zurückgesetzt.

Bei Vergleichsbefehlen wird das Bit Z auf „1" gesetzt, wenn der Vergleich Gleichheit ergibt, sonst auf „0".

Beim Z80 wird es zusätzlich für drei weitere Funktionen verwendet: Beim Befehl BIT dient es zur Anzeige des getesteten Bits. Es wird auf „1" gesetzt, wenn das getestete Bit „0" ist, sonst zurückgesetzt.

Bei den speziellen „Block-Ein/Ausgabebefehlen" (INI, IND, INIR, INDR) wird das Z-Flag gesetzt, wenn B − 1 = 0 (vor der Befehlsausführung) und sonst zurückgesetzt. Es ist gesetzt, wenn der Bytezähler auf „0" dekrementiert wird.

Schließich wird das Z-Flag bei dem speziellen Befehl IN r,(C) auf „1" gesetzt, um anzuzeigen, daß das eingegebene Byte den Wert „0" hat.

Zusammenfassend beeinflussen folgende Befehle den Wert des Bits Z: ADD A,s; ADC A,s; SUB s; SBC A,s; CP s; NEG; AND s; OR s; XOR s; INC s; DEC m; ADC HL,ss; SBC HL,ss; RL m; RLC m; RR m; RRC m; SLA m; SRA m; RLD; RRD; DAA; IN r,(C); INI; IND; INIR; INDR; OUTI; OUTD; OTIR; OTDR; CPI; CPIR; CPD; CPDR; LD A,I; LD A,R; BIT b,s; NEG.

Gebräuchliche Befehle, die das Bit Z nicht beeinflussen, sind: ADD DD,ss; RLA; RLCA; RRA; RRCA; CPL; SCF; CCF; LDI; LDD; LDIR; LDDR; INC DD; DEC DD.

Vorzeichen (Sign, S)

Dieses Flag gibt den Wert des höchsten Bits eines Ergebnisses oder eines übertragenen Bytes an (Bit sieben). In der Darstellung als Zweierkomplement wird das höchste Bit zur Angabe des Vorzeichens verwendet. „0" zeigt eine positive Zahl und „1" eine negative Zahl an. Deshalb wird Bit 7 das Vorzeichenbit genannt.

Bei den meisten Mikroprozessoren spielt das Vorzeichenbit eine wichtige Rolle, wenn man mit Ein-/Ausgabegeräten kommuniziert. Die meisten Mikroprozessoren sind nicht mit einem Befehl BIT ausgestattet, der den Inhalt eines beliebigen Bits in einem Register oder im Speicher testet. Deshalb ist das Vorzeichenbit üblicherweise am besten zum Testen geeignet. Wenn man den Status eines Ein-/Ausgabegerätes überprüft, setzt das Lesen des Statusregisters automatisch das Vorzeichen-

bit, das auf den Wert von Bit sieben des Statusregisters gesetzt wird. Es kann wie üblich im Programm getestet werden. Deshalb liegt der wichtigste Indikator (normalerweise bereit/nicht bereit) im Statusregister eines Ein-/Ausgabegerätes für Mikroprozessoren normalerweise in Bit 7.

Beim Z80 steht ein spezieller Befehl BIT zur Verfügung. Um jedoch eine Speicherstelle zu testen in der ein Ein-/Ausgabestatusregister liegen kann), muß zuerst deren Adresse in Register HL, IX oder IY geladen werden. Es gibt keinen Befehl, mit dem man ein Bit in einer festgelegten Speicherstelle direkt testen kann (d. h. keine direkte Adressierung für diesen Befehl). Die Positionierung des Flags „bereit" im Bit 7 bei Ein-/Ausgabegeräten bleibt deshalb auch beim Z80 bedeutsam.

Schließlich wird das Flag von dem Befehl IN r,(C) verwendet, um das Vorzeichen der eingegebenen Daten anzuzeigen.

Folgende Befehle beeinflussen das Vorzeichenbit: ADD A,s; SUB s; SBC A,s; CP s; NEG; AND s; OR s; XOR s; INC s; DEC m; ADC HL,ss; SBC HL,ss; RL m; RLC m; RR m; RRC m; SLA m; SRA m; SRL m; RLD; RRD; DAA; IN r,(C); CPI; CPIR; CPD; CPDR; LD A,I; LD A,R; NEG; ADC A,s.

Zusammenfassung der Flags

Die Flagbits werden dazu verwendet, spezielle Zustände innerhalb der ALU des Mikroprozessors automatisch anzuzeigen. Sie können üblicherweise mit besonderen Befehlen getestet werden, so daß abhängig von dem festgestellten Zustand bestimmte Maßnahmen ergriffen werden können. Es ist wichtig, die Funktion der verschiedenen verfügbaren Anzeiger zu verstehen, weil die meisten Entscheidungen innerhalb des Programms in Abhängigkeit vom Zustand dieser Flags getroffen werden. Alle Sprünge, die in einem Programm ausgeführt werden, werden abhängig vom Zustand dieser Flags zu bestimmten Adressen ausgeführt.

Die einzige Ausnahme bildet der Interruptmechanismus, der im Kapitel über Ein-/Ausgaben beschrieben wird, und der einen Sprung zu einer festgelegten Adresse auslösen kann, immer wenn ein Hardwaresignal an bestimmten Anschlüssen des Z80 anliegt.

An dieser Stelle ist es nur notwendig, sich die hauptsächliche Funktion der einzelnen Bits zu merken. Beim Programmieren kann der Leser die Beschreibungen der einzelnen Befehle an späterer Stelle in diesem Kapitel verwenden, um die Wirkung jedes einzelnen Befehls auf die verschiedenen Flags zu überprüfen. Meistens braucht man fast alle Flags nicht zu beachten, und der Leser, der damit noch nicht vertraut ist, braucht sich wegen der scheinbaren Kompliziertheit nicht bange machen zu lassen. Ihre Verwendung wird klarer werden, sobald wir mehr Programmanwendungen untersucht haben.

Eine Zusammenstellung der sechs Flags und die Art und Weise, wie sie von den verschiedenen Befehlen gesetzt oder zurückgesetzt werden, ist in Abbildung 4.17 gezeigt.

BEFEHL	C	Z	P/V	S	N	H	KOMMENTAR
ADD A, s; ADC A, s	↕	↕	V	↕	0	↕	8-Bit-Addition oder Addition mit Carry
SUB s; SBC A, s; CP s; NEG	↕	↕	V	↕	1	↕	8-Bit-Subtraktion, Subtraktion mit Carry, vergleiche und negiere Akkumulator
AND s	0	↕	P	↕	0	1	Logische Operationen
OR s; XOR s	0	↕	P	↕	0	0	AND setzt verschiedene Flags
INC s	•	↕	V	↕	0	↕	Inkrementiere 8 Bit
DEC m	•	↕	V	↕	1	↕	Dekrementiere 8 Bit
ADD DD, ss	↕	•	•	•	0	X	Addiere 16 Bit
ADC HL, ss	↕	↕	V	↕	0	X	Addiere 16 Bit mit Carry
SBC HL, ss	↕	↕	V	↕	1	X	Subtrahiere 16 Bitmit Carry
RLA; RLCA; RRA; RRCA	↕	•	•	•	0	0	Rotiere Akkumulator
RL m; RLC m; RR m; RRC m SLA m; SRA m; SRL m	↕	↕	P	↕	0	0	Rotiere und schiebe Stelle m
RLD, RRD	•	↕	P	↕	0	0	Rotiere Ziffer links und rechts
DAA	↕	↕	P	↕	•	↕	Gleiche Akkumulator dezimal ab
CPL	•	•	•	•	1	1	Komplementiere Akkumulator
SCF	1	•	•	•	0	0	Setze Carry
CCF	↕	•	•	•	0	X	Komplementiere Carry
IN r, (C)	•	↕	P	↕	0	0	Register-indirekte Eingabe
INI; IND; OUTI; OUTD	•	↕	X	X	1	X	Block-Eingabe und -Ausgabe
INIR; INDR; OTIR; OTDR	•	1	X	X	1	X	Z = 0, falls B = 0, sonst Z = 1
LDI, LDD	•	X	↕	X	0	0	Blocktransferbefehle
LDIR, LDDR	•	X	0	X	0	0	P/V = 1, falls BC = 0, sonst P/V = 0
CPI, CPIR, CPD, CPDR	•	↕	↕	↕	1	X	Blocksuchbefehle Z = 1, wenn A = (HL), sonst Z = 0 P/V = 1, wenn BC = 0, sonst P/V = 0
LD A, I; LD A, R	•	↕	IFF	↕	0	0	Der Inhalt des Interrupt-Freigabe-Flip-Flops (IFF) wird ins Flag P/V kopiert
BIT b, s	•	↕	X	X	0	1	Das Komplement von Bit b der Stelle s wird ins Flag Z kopiert
NEG	↕	↕	V	↕	1	↕	Negiere Akkumulator

Courtesy of Zilog, Inc.

In dieser Tabelle wurden die folgenden Abkürzungen verwendet:

SYMBOL	OPERATION
C	Übertragsflag. C=1 wenn die Operation einen Übertrag vom MSB des Operanden oder des Ergebnisses erzeugte.
Z	Nullflag. Z=1 wenn das Ergebnis der Operation Null ist.
S	Vorzeichenflag. S=1 wenn das MSB des Ergebnisses Eins ist.
P/V	Paritäs- oder Überlaufflag. Parität (P) und Überlauf (V) beeinflussen das gleiche Flag. Logische Operationen beeinflussen dieses Flag durch die Parität des Ergebnisses, während arithmetische Operationen einen Überlauf anzeigen. Enthält P/V die Parität, so ist P/V=1 bei ungerader, P/V=0 bei gerader Parität. Enthält P/V den Überlauf, so ist P/V=1, wenn ein Überlauf auftrat.
H	Halbübertragsflag. H=1 wenn Addition oder Subtraktion einen Übertrag nach Bit 4 ergaben.
N	Additions-/Subtraktionsflag. N=1 wenn die letzte Operation eine Subtraktion war. Die Flags H und N werden von dem Befehl DAA (Dezimalanpassung) verwendet, um das Ergebnis bei gepackter BCD-Darstellung nach Addition oder Subtraktion entsprechend zu korrigieren.
↕	Flag wird entsprechend dem Ergebnis der Operation beeinflußt.
•	Flag wird durch die Operation nicht beeinflußt.
0	Flag wird durch die Operation zurückgesetzt.
1	Flag wird durch die Operation gesetzt.
X	Flag ist willkürlich gesetzt.
V	Flag wird von einem Überlauf beeinflußt.
P	Flag wird durch die Parität des Ergebnisses beeinflußt.
r	Beliebiges der CPU-Register A, B, C, D, E, H, L.
s	Beliebige 8-Bit-Adresse für alle bei diesem Befehl zulässigen Adressierungsarten.
ss	Beliebige 16-Bit-Adresse für alle bei diesem Befehl zulässigen Adressierungsarten.
ii	Beliebiges Indexregister IX oder IY.
R	Auffrischzähler.
n	8-Bit-Zahl im Bereich 0 bis 255.
nn	16-Bit-Zahl im Bereich 0 bis 65535.
m	Beliebige 8-Bit-Zahl für alle bei der Operation zulässigen Adressierungsarten.

Abb. 4.17: Zusammenfassung der Arbeitsweise der Flags

Die Sprungbefehle

Eine Verzweigung ist ein Befehl, der einen Sprung zu einer festgelegten Adresse erzwingt. Er verändert den normalen Ablauf des Programms, d. h. die sequentielle Abarbeitung so, daß plötzlich ein anderer Abschnitt des Programms ausgeführt wird. Sprünge können unbedingt oder bedingt sein. Bei einem unbedingten Sprung wird unabhängig von irgendeiner anderen Bedingung zu einer festgelegten Adresse verzweigt.

Ein bedingter Sprung wird nur dann zu einer festgelegten Adresse ausgeführt, wenn eine oder mehrere Bedingungen erfüllt sind. Mit dieser Art von Sprung werden Entscheidungen in Abhängigkeit von Daten oder berechneten Ergebnisse gefällt.

Um die bedingten Sprungbefehle zu erklären, muß die Funktion der Flagregister verstanden sein, da alle Entscheidungen über Verzweigungen auf diesen Registern basieren. Dies war die Aufgabe des vorhergehenden Abschnitts. Wir können jetzt die Sprungbefehle, über die der Z80 verfügt, genauer untersuchen.

Es gibt zwei Haupttypen von Sprungbefehlen: Sprünge innerhalb des Hauptprogramms („JUMP" genannt) und die speziellen Verzweigungsbefehle, mit denen man Unterprogramme anspringt und von dort zurückkehrt („CALL" und „RETURN"). Als Ergebnis eines Sprungbefehls wird der Befehlszähler mit einer neuen Adresse geladen, und an dieser Stelle geht dann die normale Programmausführung weiter. Die ganze Leistungsfähigkeit der verschiedenen Sprungbefehle kann nur im Zusammenhang mit den verschiedenen Adressierungsarten verstanden werden, die im Mikroprozessor zur Verfügung stehen. Deshalb wird dieser Teil der Diskussion bis zum nächsten Kapitel verschoben, wo die Adressierungsarten besprochen werden. Hier wollen wir uns nur mit den anderen Gesichtspunkten dieser Befehle befassen.

Sprünge können unbedingt sein (zu einer festgelegten Speicheradresse verzweigen) oder bedingt. Bei bedingten Sprüngen kann eines von vier Flagbits getestet werden. Dies sind die Flags Z, C, P/V und S. Jedes kann auf den Wert „0" oder „1" getestet werden.

Die entsprechenden Abkürzungen sind:

Z = Null (Z=1)
NZ = nicht Null (Z=0)
C = Übertrag (C=1)
NC = kein Übertrag (C=0)
PO = ungerade Parität
PE = gerade Parität
P = plus (S=0)
M = minus (S=1)

Zusätzlich steht beim Z80 ein spezieller Kombinationsbefehl zur Verfügung, der das Register B dekrementiert und zu einer festgelegten Speicheradresse springt, solange es nicht Null ist. Dies ist ein leistungsfähiger

Befehl am Ende einer Schleife und er wurde im letzten Kapitel schon mehrfach verwendet: Es ist der Befehl DJNZ.

Entsprechend können auch die Befehle CALL und RET (RETURN) bedingt oder unbedingt sein. Sie testen die gleichen Flags wie der Sprungbefehl, den wir schon besprochen haben.

Bedingte Sprünge stellen eine sehr leistungsfähige Eigenschaft dar, über die nicht alle 8-Bit-Mikroprozessoren verfügen. Sie verbessert die Effektivität eines Programms, weil in einem Befehl untergebracht ist, wofür sonst zwei Befehle belegt werden.

Schließlich gibt es für Interruptprogramme zwei Spezialbefehle. Dies sind RETI und RETN. Sie werden in Kapitel 6 bei den Interrupts beschrieben.

Die Adressierungsarten und die Opcodes für die verschiedenen verfügbaren Verzweigungen zeigt Abb. 4.18.

Eine detaillierte Beschreibung der verschiedenen Adressierungsarten wird in Kapitel 5 gegeben.

Wenn wir Abb. 4.18 untersuchen, dann wird es offensichtlich, daß viele Adressierungsarten beschränkt sind. Der absolute Sprung JP nn kann beispielsweise vier Flags testen, während der relative Sprung JR nn nur zwei Flags testen kann.

			BEDINGUNG									
			UN. COND	CARRY	NON CARRY	ZERO	NON ZERO	PARITY EVEN	PARITY ODD	SIGN NEG	SIGN POS	REG B≠0
JUMP 'JP'	IMMED. EXT.	nn	C3 n n	DA n n	D2 n n	CA n n	C2 n n	EA n n	E2 n n	FA n n	F2 n n	
JUMP 'JR'	RELATIVE	PC+e	18 e-2	38 e-2	30 e-2	28 e-2	20 e-2					
JUMP 'JP'	REG. INDIR.	(HL)	E9									
JUMP 'JP'		(IX)	DD E9									
JUMP 'JP'		(IY)	FD E9									
'CALL'	IMMED. EXT.	nn	CD n n	DC n n	D4 n n	CC n n	C4 n n	EC n n	E4 n n	FC n n	F4 n n	
DEKREMENTIERE B, SPRINGE FALLS NICHT NULL 'DJNZ'	RELATIVE	PC+e										10 e-2
RETURN 'RET'	REGISTER INDIR.	(SP) (SP+1)	C9	D8	D0	C8	C0	E8	E0	F8	F0	
RETURN FROM INT 'RETI'	REG. INDIR.	(SP) (SP+1)	ED 4D									
RETURN FROM NON MASKABLE INT 'RETN'	REG. INDIR.	(SP) (SP+1)	ED 45									

Abb. 4.18: Sprungbefehle

Beachten Sie eine wichtige Anmerkung: Man verwendet JR fast immer dann, wenn es möglich ist, da er kürzer ist als JP (ein Byte weniger) und das Verschieben von Programmen erleichtert. JR und JP sind jedoch nicht austauschbar: JR kann die Flags Parität und Vorzeichen nicht testen.

Ein weiterer spezieller Typ von Verzweigungsbefehlen steht zur Verfügung: Der Befehl *Restart* oder RST. Er ist ein Einbytebefehl, der den Sprung zu einer von acht Adressen im unteren Teil des Speichers erlaubt. Seine Startadressen sind dezimal 0, 8, 16, 24, 32, 40, 48 und 56. Er ist ein leistungsfähiger Befehl, weil er nur aus einem Byte besteht. Er ist die schnellste Verzweigung, die zur Verfügung steht, und er ist aus diesem Grund für die Reaktion auf Interrupts wichtig. Er kann vom Programmierer aber auch zu anderen Zwecken verwendet werden. Abb. 4.19 zeigt eine Zusammenstellung der Opcodes für diesen Befehl.

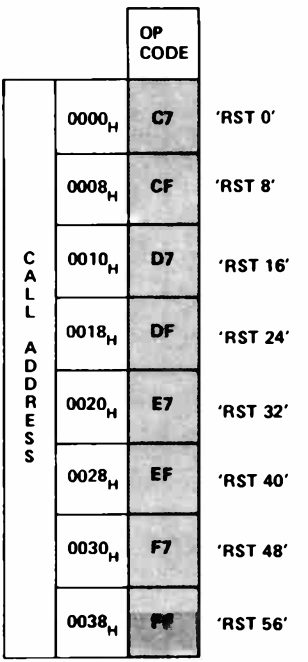


Abb. 4.19: Restart-Befehle

Ein-/Ausgabebefehle

Die Ein-/Ausgabetechniken werden in Kapitel 6 im einzelnen beschrieben. Vereinfacht gesagt kann man ein Ein-/Ausgabegerät auf zwei Ar-

ten adressieren: Als Speicherplatz, wobei man dann einen der Befehle verwendet, die schon beschrieben wurden, oder mit speziellen Ein-/Ausgabebefehlen. Die normalen Befehle zur Speicheradressierung belegen drei Byte: ein Byte für den Opcode und zwei Byte für die Adresse. Deshalb werden sie langsam ausgeführt, weil sie drei Speicheradressen benötigen. Der hauptsächliche Zweck spezieller Ein-/Ausgabebefehle ist es, kürzere und damit schnellere Befehle zu liefern. Ein-/Ausgabebefehle haben jedoch zwei Nachteile.

Erstens „verbrauchen" sie mehrere der wenigen verfügbaren Opcodes (da normalerweise nur 8 Bit für alle Opcodes zur Verfügung stehen, die in einem Mikroprozessor benötigt werden). Zweitens machen sie die Erzeugung eines oder mehrere spezieller Ein-/Ausgabesignale nötig und „verbrauchen" einen oder mehrere der wenigen Anschlüsse, die der Mikroprozessor besitzt. Die Zahl der Anschlüsse ist normalerweise auf 40 begrenzt. Wegen dieser möglichen Nachteile gibt es bei den meisten Mikroprozessoren keine speziellen Ein-/Ausgabebefehle. Es gibt sie jedoch beim 8080 (der als erster eingeführte universelle 8-Bit-Mikroprozessor) und beim Z80, der, wie wir wissen, zum 8080 kompatibel ist.

Der Vorteil von Ein-/Ausgabebefehlen ist es, daß sie schneller arbeiten, da sie nur zwei Byte belegen. Ein ähnliches Ergebnis erhält man jedoch, wenn eine spezielle Adressierungsart zur Verfügung steht, die man Adressierung in der „Seite 0" nennt, und bei der das Adressfeld auf 8 Bit begrenzt ist. Diese Lösung wurde oft bei anderen Mikroprozessoren gewählt.

Die beiden grundsätzlichen Ein-/Ausgabebefehle sind IN und OUT. Sie übertragen entweder den Inhalt einer festgelegten Eingabeadresse in eines der Arbeitsregister oder den Inhalt eines Registers zu dem Ausgabegerät. Sie sind natürlich zwei Byte lang. Das erste Byte ist für den Opcode reserviert, das zweite Byte des Befehls bildet den unteren Teil der Adresse. Auf der oberen Hälfte des Adreßbusses wird der Akkumulator ausgegeben. So könnte man eines von 64K Geräten auswählen. Da dies jedoch Zeit kostet (der Akkumulator muß geladen werden) und außerdem der Akkumulator normalerweise Daten enthält, die ein- bzw. ausgegeben werden, selektiert man mit den Ein-/Ausgabebefehlen üblicherweise nur 256 Geräte.

Zusätzlich verfügt der Z80 über eine Register-indirekte Betriebsart und über vier spezielle Befehle zur Ein-/Ausgabe von Blöcken.

Bei dem Befehl Registereingabe, der das Format IN r,(C) hat, dient das Registerpaar BC (bzw. das Register C, wenn man wie üblich maximal 256 Geräte adressiert) als Zeiger auf das Ein-/Ausgabegerät. Der Inhalt des Registers B wird auf dem oberen Teil des Adreßbusses ausgegeben. Der Inhalt des ausgewählten Ein-/Ausgabegerätes wird dann in das mit r festgelegte Register geladen.

Das gleiche gilt auch für den Befehl OUT.

Die vier Blocktransferbefehle sind für die Eingabe: INI, INIR (INI wiederholt), IND und INDR (IND wiederholt). Für die Ausgabe sind dies entsprechend: OUTI, OTIR, OUTD und OTDR.

Bei diesem automatischen Blocktransfer dient HL als Zeiger in den Speicher. Das Register C selektiert das Ein-/Ausgabegerät (eines von 256). Bei einem Ausgabebefehl zeigen H und L auf die Quelle. Das Register B dient als Zähler und wird dekrementiert. Bei der Eingabe sind die entsprechenden Befehle INI, wobei HL inkrementiert wird, und IND, wobei HL dekrementiert wird.

INI ist ein automatischer Einbytetransfer. Register C selektiert das Ein-/Ausgabegerät. Es wird ein Byte aus dem Ein-/Ausgabebaustein gelesen und in der Speicherstelle abgelegt, auf die H und L zeigen. Dann werden H und L um 1 inkrementiert und B um 1 dekrementiert.

INIR ist der gleiche Befehl, aber automatisiert. Er wird wiederholt ausgeführt, bis der Zähler B auf Null dekrementiert wird. So können bis zu 256 Byte automatisch übertragen werden. Beachten Sie, daß man vor der Ausführung des Befehls das Register B auf „0" setzen muß, wenn man genau 256 Byte übertragen will.

Die Opcodes für die Ein-/Ausgabebefehle sind in Abb. 4.20 und 4.21 zusammengefaßt.

QUELLE

			REGISTER							REG. IND.	
			A	B	C	D	E	H	L	(HL)	
'OUT'	IMMED.	(n)	D3 n								
	REG. IND.	(C)	ED 79	ED 41	ED 49	ED 51	ED 59	ED 61	ED 69		
'OUTI' – OUTPUT Inc HL, Dec b	REG. IND.	(C)								ED A3	BLOCK-AUSGABE-BEFEHLE
'OTIR' – OUTPUT, Inc HL, Dec B, REPEAT IF B≠0	REG. IND.	(C)								ED B3	
'OUTD' – OUTPUT Dec HL & B	REG. IND.	(C)								ED AB	
'OTDR' – OUTPUT, Dec HL & B, REPEAT IF B≠0	REG. IND.	(C)								ED BB	

ADRESSE DES ZIELPORTS

Abb. 4.20: Die Ausgabebefehle

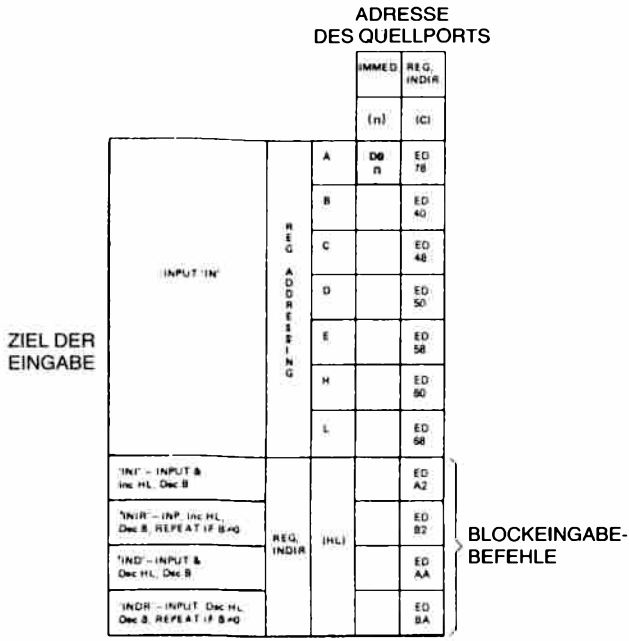


Abb. 4.21: Die Eingabebefehle

Steuerbefehle

Steuerbefehle sind Befehle, die die Betriebsart der CPU verändern oder ihre interne Statusinformation beeinflussen. Es gibt sieben solche Befehle.

Der Befehl NOP ist ein funktionsloser Befehl, der einen Zyklus lang nichts tut. Er wird typischerweise benutzt, um eine beabsichtigte Verzögerung (4 Zustände = 2 Mikrosekunden bei 2 MHz Takt) zu bewirken, oder um die Löcher zu füllen, die bei der Fehlersuche und Korrektur entstehen. Um die Fehlersuche in einem Programm zu erleichtern, besteht der Opcode für NOP traditionell aus lauter Nullen. Dies ist der Fall, weil zur Ausführungszeit der Speicher oft gelöscht ist, d. h. lauter Nullen enthält. Werden NOPs ausgeführt, dann wird weder etwas zerstört noch die Programmausführung gestoppt.

Der Befehl HALT wird in Verbindung mit Interrupts oder einem Reset benutzt. Er unterbricht tatsächlich die Operation der CPU. Die CPU wird die Arbeit wieder aufnehmen, sobald ein Interrupt oder ein Reset auftritt. In dieser Betriebsart führt die CPU ständig NOPs aus. Während der Fehlersuche plaziert man oft ein HALT am Ende des Programms, da das Hauptprogramm üblicherweise nichts anderes tun muß. Das Programm muß dann ausdrücklich neu gestartet werden.

Zwei spezielle Befehle dienen dazu, das Interruptflag ein- bzw. auszuschalten. Dies sind EI und DI. Interrupts werden im Kapitel 6 beschrieben. Das Interruptflag dient dazu, Interrupts zuzulassen oder zu sperren. Wenn man verhindern will, daß Interrupts während spezieller Teile eines Programms auftreten, kann man das Interrupt-Flip-Flop mit diesem Befehl sperren. Er wird in Kapitel 6 angewendet. Diese Befehle zeigt Abb. 4.22.

'NOP'	00	
'HALT'	76	
DISABLE INT '(DI)'	F3	
ENABLE INT '(EI)'	FB	
SET INT MODE 0 'IM0'	ED 46	8080 MODUS
SET INT MODE 1 'IM1'	ED 56	AUFRUF ZUR ADRESSE 0038_H
SET INT MODE 2 'IM2'	ED 5E	INDIREKTER AUFRUF MIT DEM REGISTER I und 8 BITS VOM UNTERBRECHENDEN GERÄT ALS ZEIGER

Abb. 4.22: Verschiedene CPU Steuerbefehle

Schließlich besitzt der Z80 drei verschiedene Interrupt-Betriebsarten. (Beim 8080 gibt es nur eine.) Die Interrupt-Betriebsart 0 ist die Betriebsart des 8080, die Betriebsart 1 ist ein Call auf die Adresse 038H und die Betriebsart 2 ist ein indirekter Aufruf, der den Inhalt des Spezialregisters I sowie 8 Bit, die das Gerät liefert, das den Interrupt auslöst, als Zeiger auf die Interrupt-Routine im Speicher verwendet. Diese Betriebsarten werden in Kapitel 6 erklärt.

Schließlich wird der Interrupt-Mechanismus beim Z80 über zwei spezielle Anschlüsse ausgelöst, die auch im Kapitel 6 erklärt werden. Dies sind die Anschlüsse INT und NMI.

Zusammenfassung

Die fünf Befehlsklassen, die beim Z80 verfügbar sind, wurden jetzt beschrieben. Die Einzelheiten der speziellen Befehle werden im folgenden Abschnitt des Buchs beschrieben. Es ist nicht notwendig, die Funktion

jedes einzelnen Befehls zu kennen, wenn man mit dem Programmieren beginnt. Am Anfang genügt die Kenntnis einiger weniger Befehle aus jeder Gruppe. Wenn Sie jedoch eigene Programme schreiben, sollten Sie alle Z80-Befehle kennen, um gute Programme zu schreiben. Am Anfang ist natürlich die Effektivität nicht so wichtig und deshalb kann man die meisten Befehle noch übergehen.

Ein wichtiger Gesichtspunkt ist noch nicht beschrieben worden. Dies ist der Satz Adressierungsarten, die im Z80 eingebaut sind, um die Verwaltung der Daten im Speicherbereich zu erleichtern. Diese Adressierungstechniken werden im nächsten Kapitel beschrieben.

Die Z80-Befehle: Einzelbeschreibungen

Abkürzungen

FLAG	EIN	AUS
Übertrag	C (carry)	NC (no carry)
Vorzeichen	M (minus)	P (plus)
Null	Z (zero)	NZ (non zero)
Parity	PE (even, gerade)	PO (odd, ungerade)

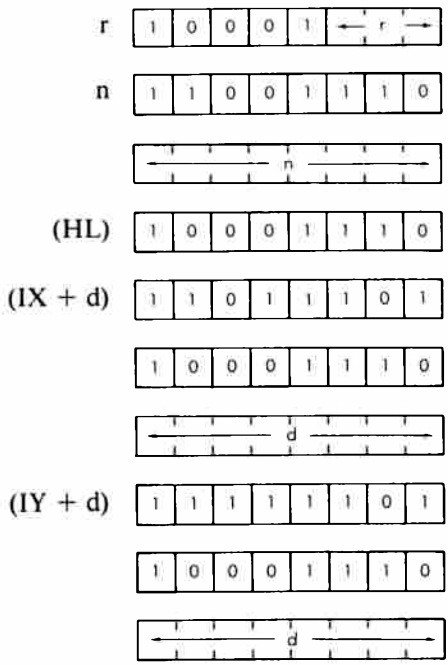
● Flag wurde entsprechend der Operation geändert.
0 Flag ist Null.
1 Flag istEins.
? Flag ist nach der Operation unbestimmt.
X Spezialfall, siehe Anmerkung auf dieser Seite.

Die Stellen 3 und 5 sind immer zufällig.

ADC A, s Addiere den Akkumulator und den festgelegten Operanden mit Übertrag (Carry).

Funktion: A ← A + s + C

Format: s: kann sein r, n, (HL),(IX + d), or (IY + d)

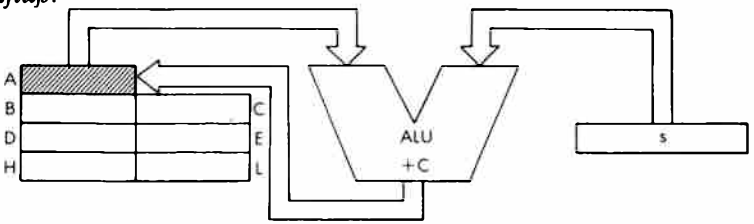


r kann sein:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Beschreibung: Der Operand s und das Übertragsflag C aus dem Statusregister werden zum Akkumulator addiert und das Ergebnis im Akkumulator gespeichert. s ist bei der Beschreibung des ähnlichen Befehls ADD definiert.

Datenfluß:



Befehls-ausführung:

s:	*M Zyklen*	*T Zustände*	*µsek @ 2 MHz:*
r	1	4	2
n	2	7	3.5
(HL)	2	7	3.5
(IX + d)	5	19	9.5
(IY + d)	5	19	9.5

Adressierungsart: r: implizit; n: unmittelbar; (HL): indirekt; (IX + d),(IY + d): indiziert.

Byte Kode: ADC A,r

r:	A	B	C	D	E	H	L
	8F	88	89	8A	8B	8C	8D

Flags:

S	Z		H		P/V	N	C
●	●		●		●	○	●

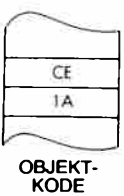
Beispiel: ADC A, 1A

Vorher:

A	06	13	F

Nachher:

A	21	11	F



ADC HL, ss Addiere HL und das Registerpaar ss mit Übertrag (Carry).

Funktion: HL ← HL + ss + C

Format:

1	1	1	0	1	1	0	1

Byte 1: ED

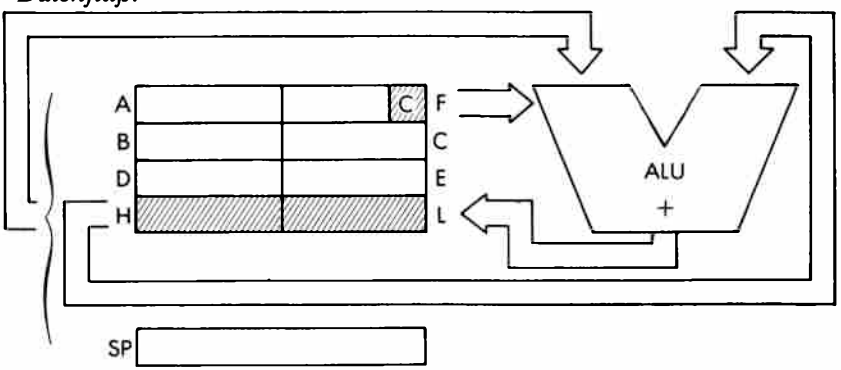
0	1	s	s	1	0	1	0

Byte 2

Beschreibung: Der Inhalt des Registerpaares HL und der Inhalt des angegebenen Registerpaares werden addiert, dann wird noch der Inhalt des Übertragsflags addiert. Das Endergebnis wird wieder in HL gespeichert. ss kann sein:

BC – 00 HL – 10
DE – 01 SP – 11

Datenfluß:



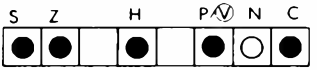
Befehlsausführung: 4 M Zyklen; 15 T Zustände: 75 µsek @ 2 MHz

Adressierungsart: Implizit.

Byte Kode:

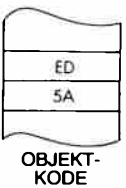
ss:	BC	DE	HL	SP
ED–	4A	5A	6A	7A

Flags:

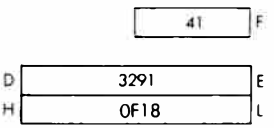


H ist gesetzt, wenn ein Übertrag von Bit 11 auftritt.

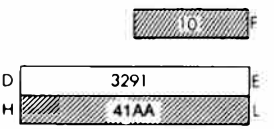
Beispiel: ADC HL, DE



Vorher:



Nachher:



ADD A, (HL) Addiere den Akkumulator mit der indirekt adressierten Speicherstelle (HL).

Funktion: A ← A + (HL)

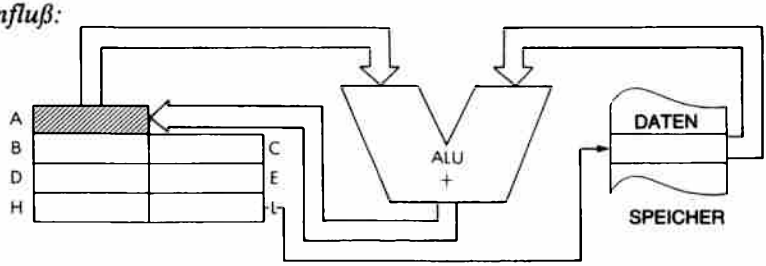
Format:

1	0	0	0	0	1	1	0

86

Beschreibung: Der Inhalt des Akkumulators wird mit dem Inhalt der Speicherstelle addiert, deren Adresse im Registerpaar HL steht. Das Ergebnis wird im Akkumulator gespeichert.

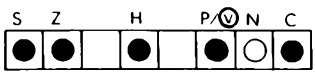
Datenfluß:



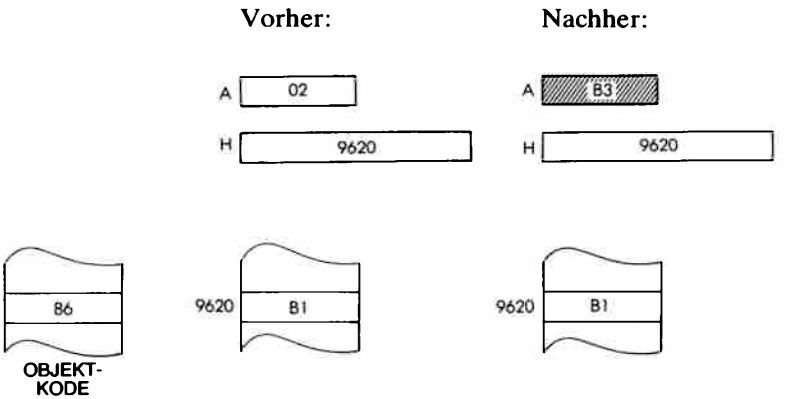
Befehlsablauf: 2 M Zyklen; 7 T Zustände: 3,5 µsek @ 2 MHz

Adressierungsart: Indirekt.

Flags:



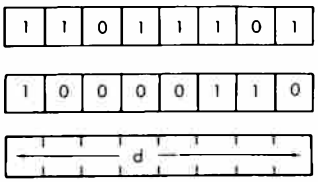
Beispiel: ADD A, (HL)



ADD A, (IX + d) Addiere den Akkumulator mit der indiziert adressierten Speicherstelle (IX + d)

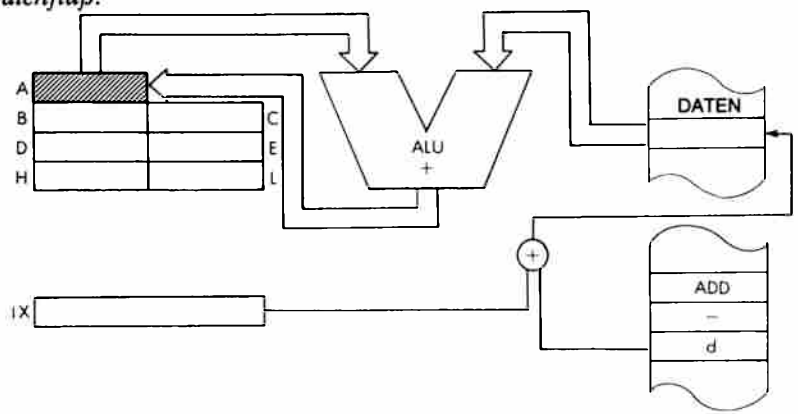
Funktion: A ← A + (IX + d)

Format:



Beschreibung: Der Inhalt des Akkumulators und der Inhalt der Speicherstelle werden addiert, die durch den Inhalt des Registers IX plus den unmittelbaren Offset adressiert wird. Das Ergebnis wird im Akkumulator gespeichert.

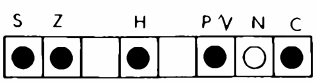
Datenfluß:



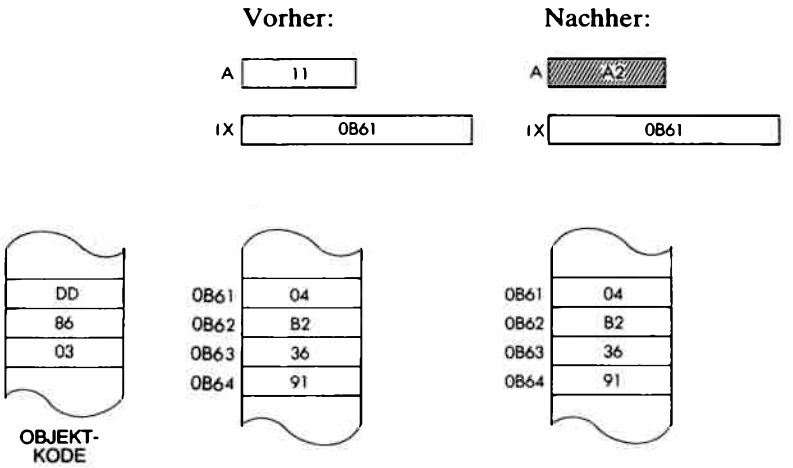
Befehlsablauf: 5 M Zyklen; 19 T Zustände: 9,5 µsek @ 2 MHz

Adressierungsart: Indiziert.

Flags:



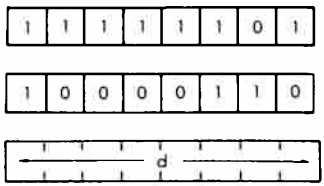
Beispiel: ADD A, (IX + 3)



ADD A, (IY + d) Addiere den Akkumulator mit der indiziert adressierten Speicherstelle (IY + d)

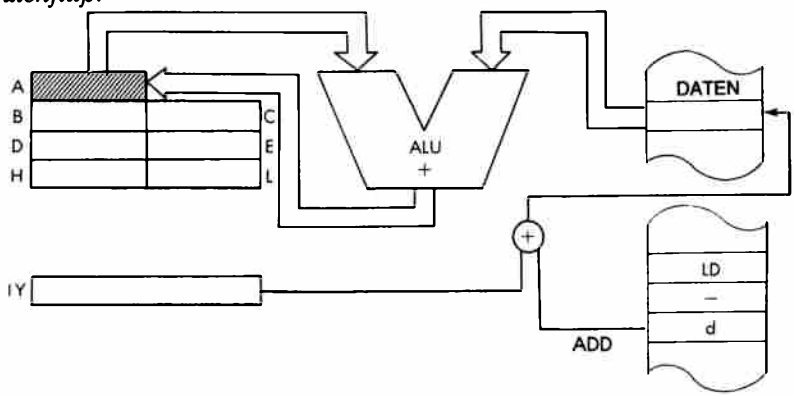
Funktion: A ← A + (IY + d)

Format:



Beschreibung: Der Inhalt des Akkumulators und der Inhalt der Speicherstelle werden addiert, die durch den Inhalt des Registers IY plus den unmittelbaren Offset adressiert wird. Das Ergebnis wird im Akkumulator abgelegt.

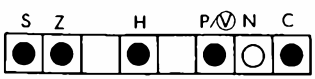
Datenfluß:



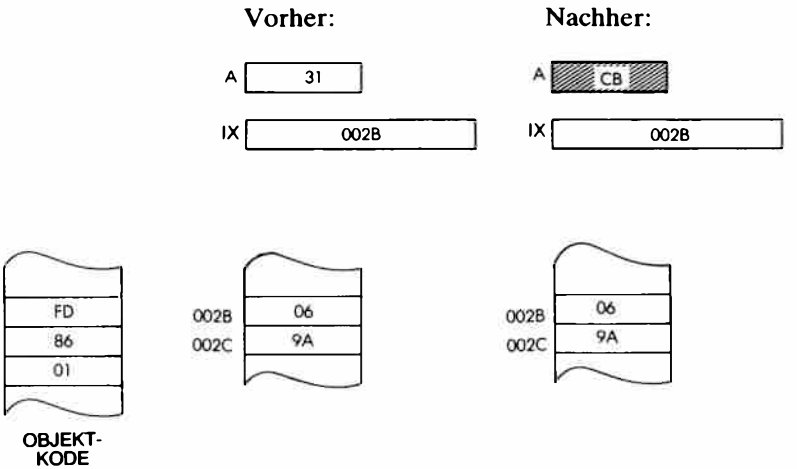
Befehlsablauf: 5 M Zyklen; 19 T Zustände: 9,5 µsek @ 2 MHz

Adressierungsart: Indiziert.

Flags:



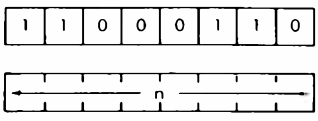
Beispiel: ADD A, (IY + 1)



ADD A, n Addiere den Akkumulator mit den unmittelbaren Daten n.

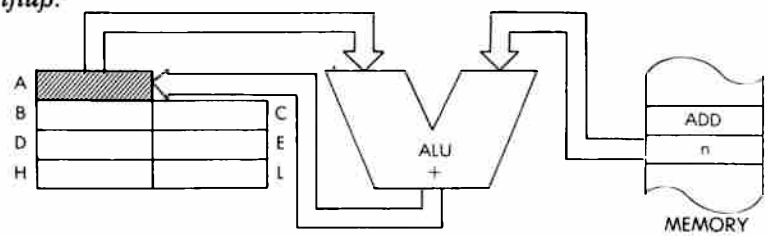
Funktion: A ← A + n

Format:



Beschreibung: Der Inhalt des Akkumulators und der Inhalt der Speicherzelle, die unmittelbar auf den Opcode folgt, werden addiert. Das Ergebnis wird im Akkumulator abgelegt.

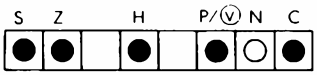
Datenfluß:



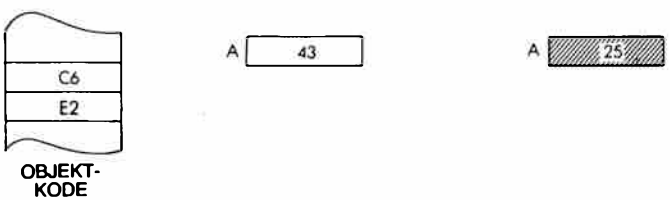
Befehlsablauf: 2 M Zyklen; 7 T Zustände: 3,5 µsek @ 2 MHz

Adressierungsart: Unmittelbar.

Flags:



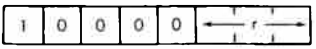
Beispiel: ADD A, E2



ADD A, r Addiere den Akkumulator mit dem Register r.

Funktion: A ← A + r

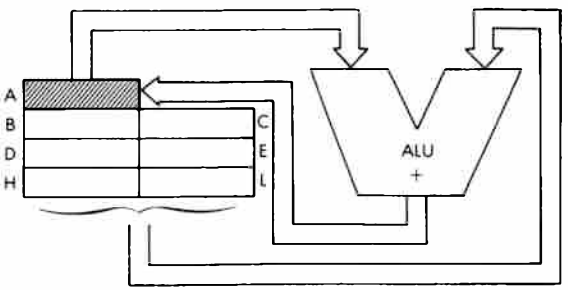
Format:



Beschreibung: Der Inhalt des Akkumulators wird mit dem Inhalt des angegebenen Registers addiert. Das Ergebnis wird im Akkumulator abgelegt. r kann sein:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Datenfluß:



Befehlsablauf: 1 M Zyklen; 4 T Zustände: 2 µsek @ 2 MHz

Adressierungsart: Implizit.

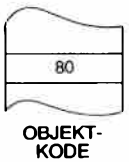
Byte Kode: r:

A	B	C	D	E	H	L
87	80	81	82	83	84	85

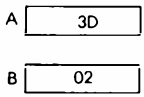
Flags:

S	Z		H		P/V	N	C
●	●		●		●	○	●

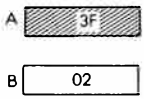
Beispiel: ADD A, B



Vorher:



Nachher:



ADD HL, ss Addiere HL und das Registerpaar ss.

Funktion: HL ← HL + ss

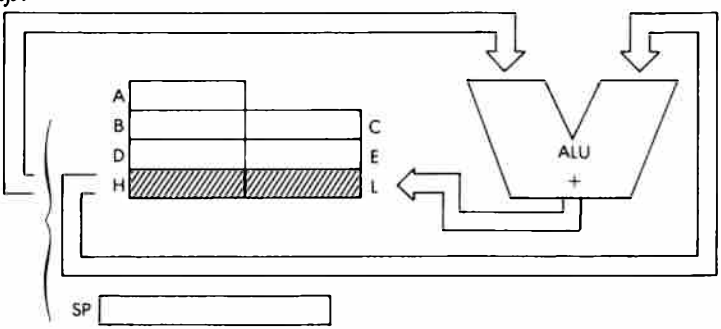
Format:

0	0	S	S	1	0	0	1

Beschreibung: Der Inhalt des angegebenen Registerpaares wird zum Inhalt des Registerpaares HL addiert und das Ergebnis in HL gespeichert. ss kann sein:

BC – 00 HL – 10
DE – 01 SP – 11

Datenfluß:



Befehlsablauf: 3 M Zyklen; 11 T Zustände: 5,5 µsek @ 2 MHz

Adressierungsart: Implizit.

Byte Kode: ss:

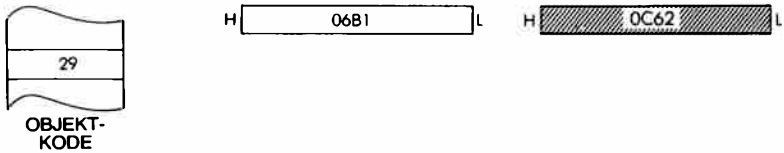
BC	DE	HL	SP
09	19	29	39

Flags:

S	Z		H		P/V	N	C
			●			○	●

C wird durch einen Übertrag vom Bit 15 gesetzt, sonst zurückgesetzt. H wird durch einen Übertrag von Bit 11 gesetzt.

Beispiel: ADD HL, HL



ADD IX, rr

Addiere IX mit dem Registerpaar rr.

Funktion: IX ← IX + rr

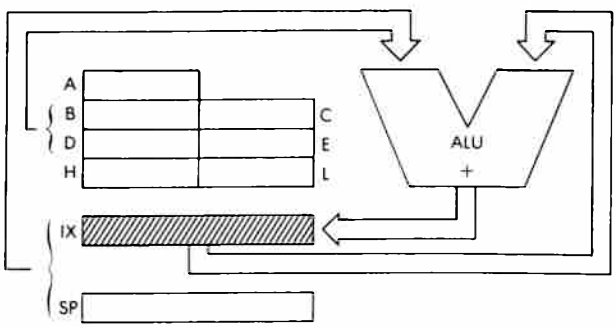
Format:

1	1	0	1	1	1	0	1	Byte 1: DD
0	0	r	r	1	0	0	1	Byte 2

Beschreibung: Der Inhalt des Registers IX wird mit dem Inhalt des angegebenen Registerpaares addiert und das Ergebnis in IX abgelegt. rr kann sein:

BC – 00	IX – 10
DE – 01	SP – 11

Datenfluß:



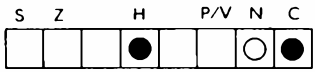
Befehlsablauf: 4 M Zyklen; 15 T Zustände: 7,5 μsek @ 2 MHz

Adressierungsart: Implizit.

Byte Kode:

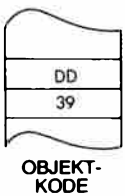
rr:	BC	DE	IX	SP
DD–	09	19	29	39

Flags:

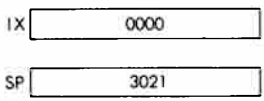


H wird gesetzt durch einen Übertrag von Bit 11.
C wird gesetzt durch einen Übertrag von Bit 15.

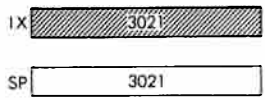
Beispiel: ADD IX, SP



Vorher:



Nachher:



ADD IY, rr

Addiere IY und das Registerpaar rr.

Funktion: IY ← IY + rr

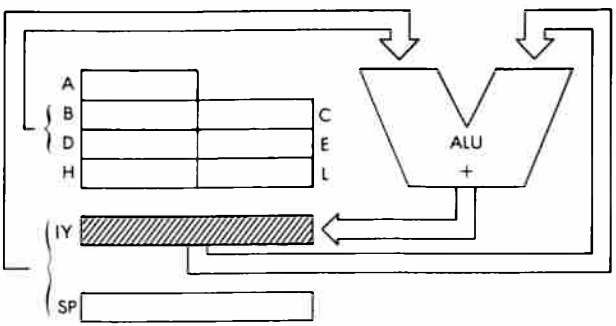
Format:

1	1	1	1	1	1	0	1	Byte 1: FD
0	0	r	r	1	0	0	1	Byte 2

Beschreibung: Der Inhalt des Registers IY wird mit dem Inhalt des angegebenen Registerpaares addiert und das Ergebnis in IY abgelegt. rr kann sein:

BC – 00	IY – 10
DE – 01	SP – 11

Datenfluß:



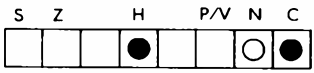
Befehlsablauf: 4 M Zyklen; 15 T Zustände: 7,5 µsek @ 2 MHz

Adressierungsart: Implizit.

Byte Kode:

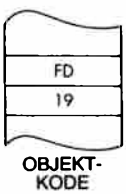
rr:	BC	DE	IY	SP
FD–	09	19	29	39

Flags:

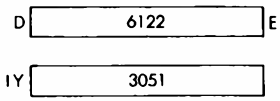


H wird durch einen Übertrag von Bit 11 gesetzt.
C wird durch einen Übertrag von Bit 15 gesetzt.

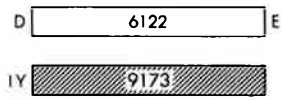
Beispiel: ADD IY, DE



Vorher:



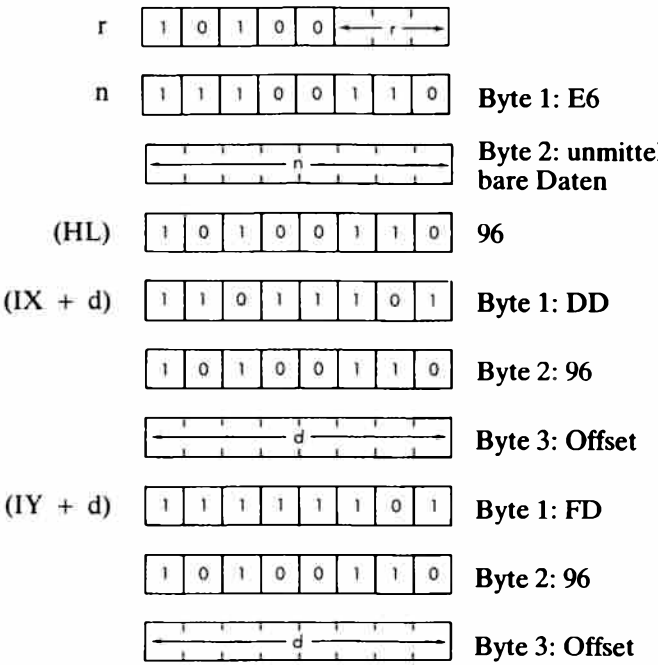
Nachher:



AND s Akkumulator und Operand s werden durch die logische Funktion „UND" verknüpft.

Funktion: $A \leftarrow A \wedge s$

Format: s: kann sein r,n,(HL),(IX + d), oder (IY + d)

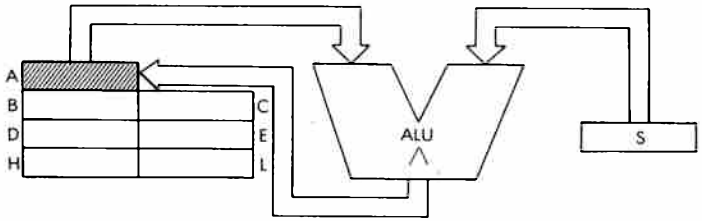


r kann sein:

A – 111	E – 011
B – 000	H – 100
C – 011	L – 101
D – 010	

Beschreibung: Der Akkumulator und der angegebene Operand werden logisch „UND" verknüpft und das Ergebnis im Akkumulator abgelegt. s ist bei der Beschreibung des ähnlichen Befehls ADD definiert.

Datenfluß:



Befehlsablauf:

s:	M Zyklen:	T Zustände:	µsek @ 2 MHz:
r	1	4	2
n	2	7	3.5
(HL)	2	7	3.5
(IX + d)	5	19	9.5
(IY + d)	5	19	9.5

Adressierungsart: r: implizit; n: unmittelbar; (HL): indirekt; (IX + d), (IY + d): indiziert.

Byte Kode: AND r

r:	A	B	C	D	E	H	L
	A7	A0	A1	A2	A3	A4	A5

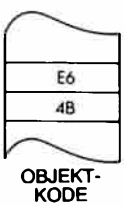
Flags:

S	Z		H		P/V	N	C
●	●		1		●	○	○

Beispiel: AND 4B

Vorher: A 36

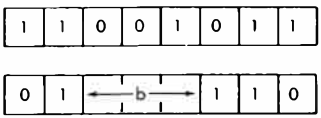
Nachher: A 02



BIT b, (HL) Teste Bit b der indirekt adressierten Speicherzelle (HL)

Funktion: $Z \leftarrow \overline{(HL)_b}$

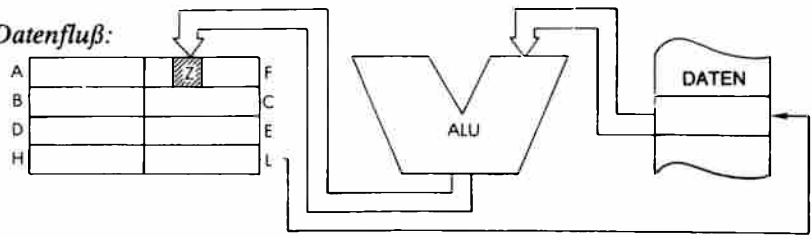
Format:



Beschreibung: Das angegebene Bit der Speicherstelle, die durch den Inhalt des Registerpaares HL adressiert wird, wird getestet, und das Flag Z entsprechend dem Ergebnis gesetzt. b kann sein:

0 – 000	4 – 100
1 – 001	5 – 101
2 – 010	6 – 110
3 – 011	7 111

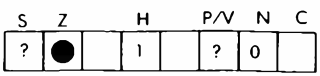
Datenfluß:



Befehlsablauf: 3 M Zyklen; 12 T Zustände: 6 µsek @ 2 MHz

Adressierungsart: Indirekt.

Flags:

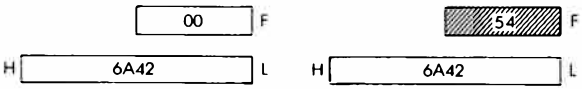


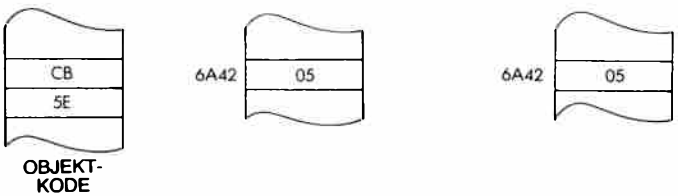
Byte Kode:

b:	0	1	2	3	4	5	6	7
CB-	46	4E	56	5E	66	6E	76	7E

Beispiel: BIT 3, (HL)

Vorher: **Nachher:**

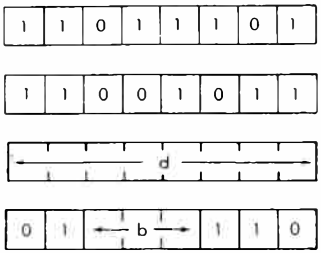




BIT b, (IX + d) Teste Bit b der indiziert adressierten Speicherzelle (IX + d)

Funktion: $Z \leftarrow (IX + d)_b$

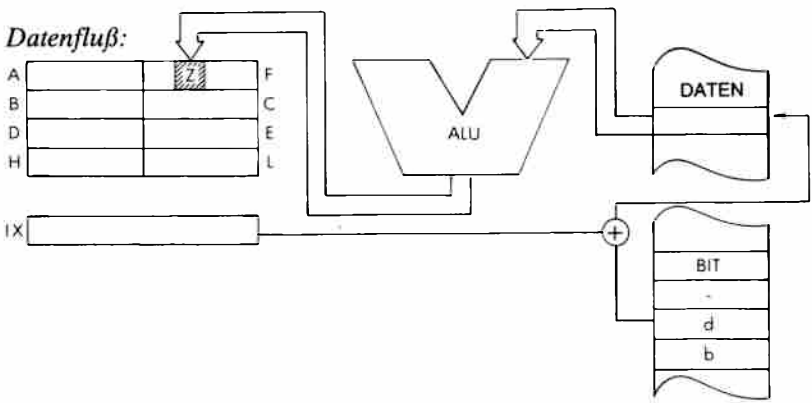
Format:



Beschreibung: Das angegebene Bit der Speicherzelle, die durch den Inhalt des Registers IX plus einem gegebenen Offset adressiert wird, wird getestet, und das Flag Z entsprechend dem Ergebnis gesetzt. b kann sein:

0 – 000	5 – 101
1 – 001	6 – 110
2 – 010	7 – 111
3 – 011	
4 – 100	

Datenfluß:



Befehlsablauf: 5 M Zyklen; 20 T Zustände: 10 μsek @ 2 MHz

Adressierungsart: Indiziert.

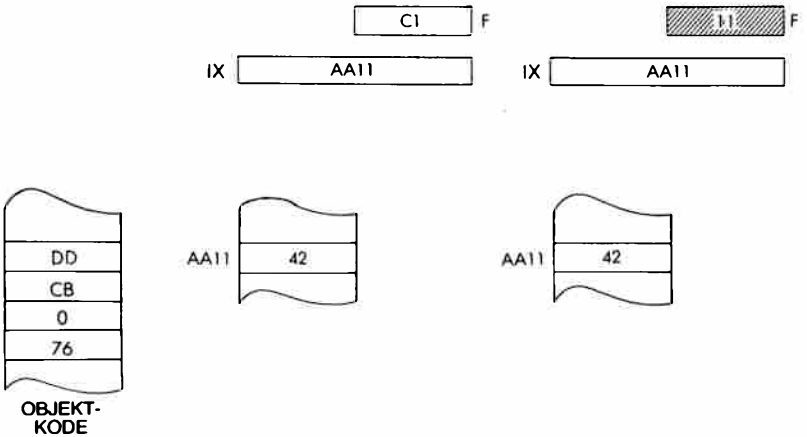
Byte Kode:

b:	0	1	2	3	4	5	6	7
DD-CB-d-	46	4E	56	5E	66	6E	76	7E

Flags:

S	Z		H		P/V	N	C
?	●		1		?	0	

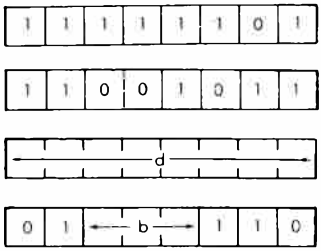
Beispiel: BIT 6, (IX + 0)



BIT b, (IY + d) Teste Bit b der indiziert adressierten Speicherstelle (IY + d)

Funktion: $Z \leftarrow \overline{(IY + d)_b}$

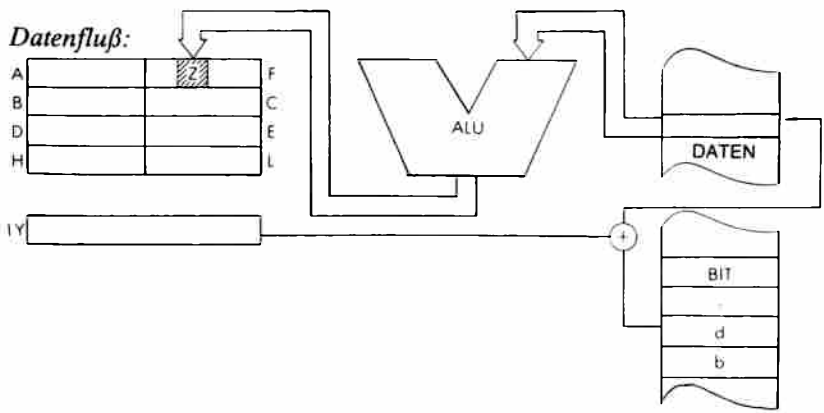
Format:



Beschreibung: Das angegebene Bit der Speicherstelle, die durch den Inhalt des Registers IY plus einem gegebenen Offset adressiert wird, wird getestet, und das Flag Z entsprechend dem Ergebnis gesetzt. b kann sein:

0 – 000	4 – 100
1 – 001	5 – 101
2 – 010	6 – 110
3 – 011	

Datenfluß:



Befehlsablauf: 5 M Zyklen; 20 T Zustände: 10 μsek @ 2 MHz

Adressierungsart: Indiziert.

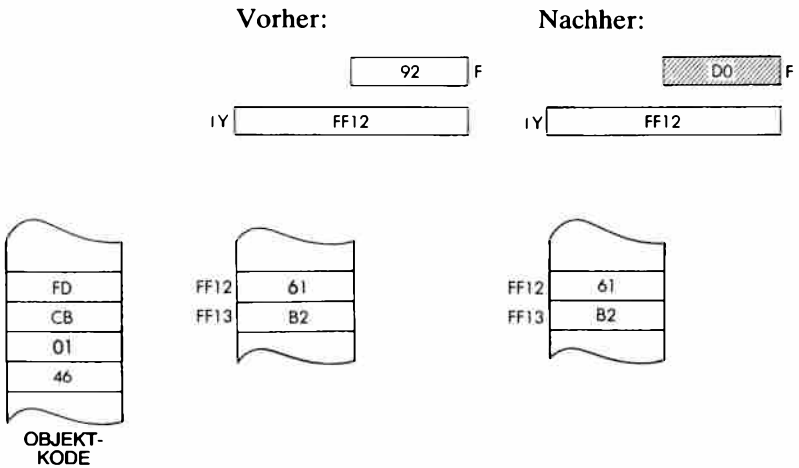
Byte Kode: b:

0	1	2	3	4	5	6	7
46	4E	56	5E	66	6E	76	7E

Flags:

S	Z		H		P/V	N	C
?	●		1		?	0	

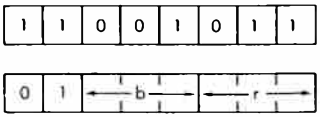
Beispiel: BIT 0, (IY + 1)



BIT b, r Teste Bit b des Registers r.

Funktion: $Z \leftarrow \overline{r_b}$

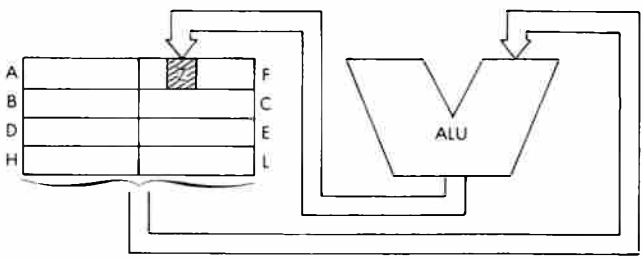
Format:



Beschreibung: Das angegebene Bit des festgelegten Registers wird getestet und das Flag Z entsprechend dem Ergebnis gesetzt. b und r können sein:

b:	0 – 000	4 – 100
	1 – 001	5 – 101
	2 – 010	6 – 110
	3 – 011	7 – 111
r:	A – 111	E – 011
	B – 000	H – 100
	C – 001	L – 101
	D – 010	

Datenfluß:



Befehlsablauf: 2 M Zyklen; 8 T Zustände: 4 μsek @ 2 MHz

Adressierungsart: Implizit.

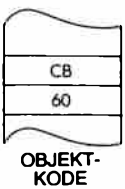
Byte Kode:

b: r:	A	B	C	D	E	H	L
0	47	40	41	42	43	44	45
1	4F	48	49	4A	4B	4C	4D
2	57	50	51	52	53	54	55
3	5F	58	59	5A	5B	5C	5D
4	67	60	61	62	63	64	65
5	6F	68	69	6A	6B	6C	6D
6	77	70	71	72	73	74	75
7	7F	78	79	7A	7B	7C	7D

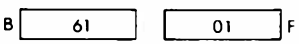
Flags:

S	Z		H		P/V	N	C
?	●		1		?	0	

Beispiel: **BIT 4, B**



Vorher:



Nachher:

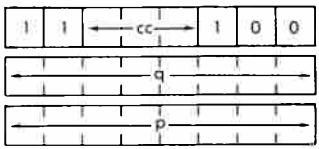
CALL cc, pq

Bedingter Unterprogrammaufruf.

Funktion:

wenn cc erfüllt (SP − 1) ← PC_{oben}; (SP − 2) ← PC_{unten}; SP ← SP − 2; PC ~ pq
Wenn cc nicht erfüllt: PC ← PC + 3

Format:



Byte 1
Byte 2: Adresse, unterer Teil
Byte 3: Adresse, oberer Teil

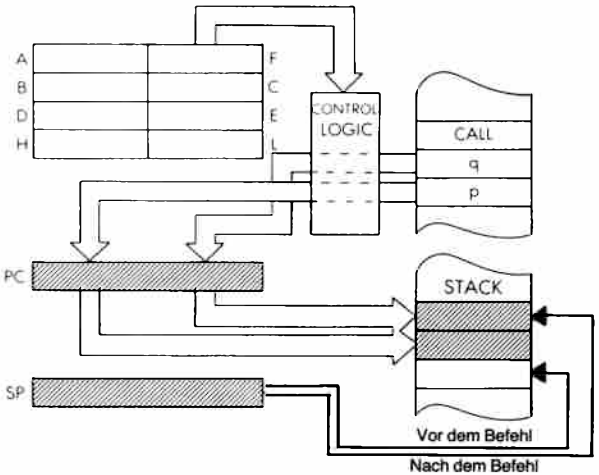
Beschreibung:

Ist die Bedingung erfüllt, dann wird der Inhalt des Befehlszählers auf den Stapel abgelegt, wie bei dem Befehl PUSH beschrieben. Dann wird der Inhalt der Speicherzelle, die unmittelbar auf den Opcode folgt, in die untere Hälfte von PC geladen, und der Inhalt der zweiten Speicherzelle nach dem Opcode in die obere Hälfte von PC. Der nächste Befehl wird dann von dieser neuen Adresse geholt. Ist die Bedingung nicht erfüllt, dann wird die Adresse pq ignoriert und der folgende Befehl ausgeführt. cc kann sein:

NZ − 000	PO − 100
Z − 001	PE − 101
NC − 010	P − 100
C − 011	M − 111

Mit dem Befehl RET kann der Inhalt von PC am Ende des aufgerufenen Unterprogramms wiederhergestellt werden.

Datenfluß:



Befehlsablauf:

	M Zyklen:	*T Zustände:*	*µsek @ 2 MHz*
Bedingung erfüllt:	5	17	8.5
Bedingung nicht erfüllt:	3	10	5

Adressierungsart: **Unmittelbar.**

Byte Kode:

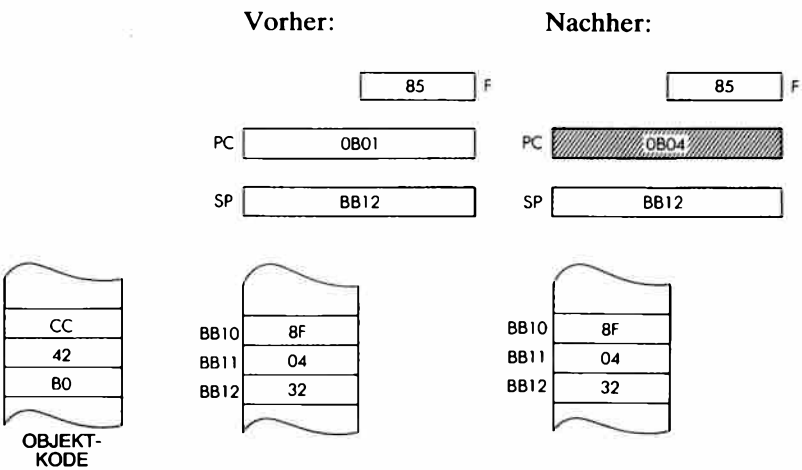
CC:	NZ	Z	NC	C	PO	PE	P	M	
C4	CC	D4	DC	E4	EC	F4	FC		q-p

Flags:

S	Z		H		P/V	N	C

(kein Einfluß)

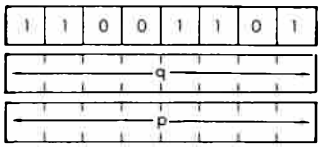
Beispiel: CALL Z,B042



CALL pq Aufruf eines Unterprogramms an der Adresse pq.

Funktion: $(SP - 1) \leftarrow PC_{oben}$; $(SP - 2) \leftarrow PC_{unten}$; $SP \leftarrow SP - 2$; $PC \leftarrow pq$

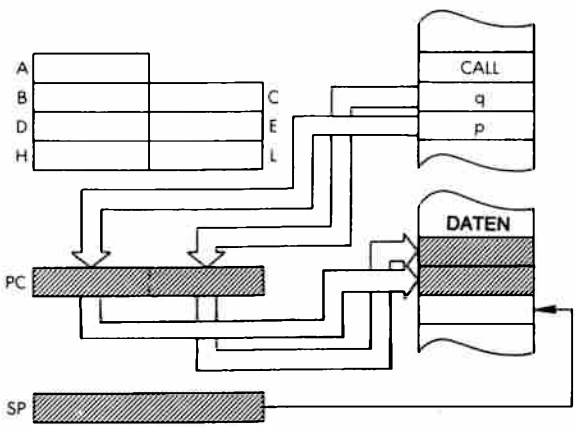
Format:



Byte 1: CD
Byte 2: Adresse, untere Hälfte
Byte 3: Adresse, obere Hälfte

Beschreibung: Der Inhalt des Befehlszählers wird auf dem Stapel abgelegt, wie bei dem Befehl PUSH beschrieben. Der Inhalt der Speicherzelle, die unmittelbar auf den Opcode folgt, wird in die untere Hälfte, der Inhalt der darauf folgenden Speicherzelle in die obere Hälfte des Befehlszählers geladen. Der nächste Befehl wird von dieser Adresse geholt.

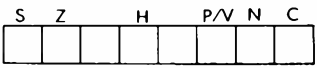
Datenfluß:



Befehlsablauf: 5 M Zyklen; 17 T Zustände: 8,5 µsek @ 2 MHz

Adressierungsart: Unmittelbar.

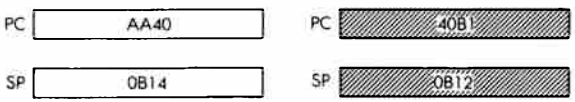
Flags:

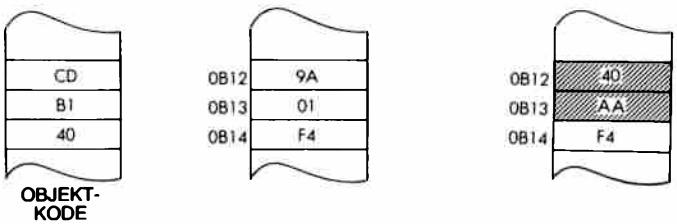


(kein Einfluß)

Beispiel: CALL 40B1

Vorher: **Nachher:**





CCF Komplementiere Übertragsflag.

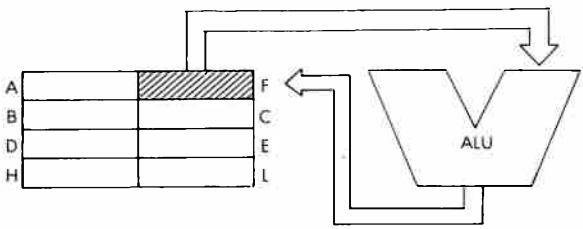
Funktion: $C \leftarrow \overline{C}$

Format:

0	0	1	1	1	1	1	1	3F

Beschreibung: Das Übertragsflag wird komplementiert.

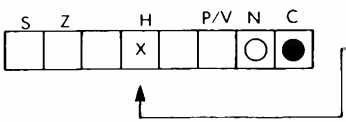
Datenfluß:



Befehlsablauf: 1 M Zyklen; 4 T Zustände: 2 μsek @ 2 MHz

Adressierungsart: Implizit.

Flags:

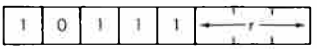


Enthält den vorhergehenden Status des Übertragsflags.

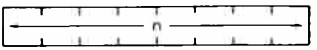
CP s Vergleiche den Operanden s mit dem Akkumulator.

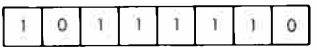
Funktion: A − s

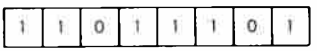
Format: s: kann sein r,n,(HL),(IX + d), oder (IY + d)

r 

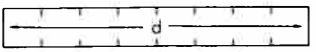
n FE

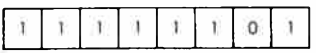
 Byte 2: unmittelbare Daten

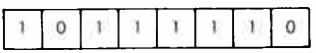
(HL)  Byte 1: BE

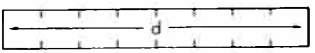
(IX + d)  Byte 1: DD

Byte 2: BE

 Byte 3: Offset

(IY + d)  Byte 1: FD

 Byte 2: BE

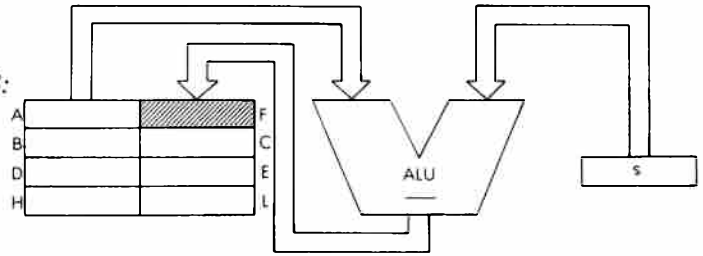
 Byte 3: Offset

r kann sein:

A − 111	E − 011
B − 000	H − 100
C − 001	L − 101
D − 010	

Beschreibung: Der angegebene Operand wird vom Akkumulator subtrahiert und das Ergebnis wird nicht weiter berücksichtigt. s ist bei der Beschreibung des Befehls ADD definiert.

Datenfluß:



Befehlsablauf:

s:	M Zyklen:	T Zustände:	µsek @ 2 MHz:
r	1	4	2
n	2	7	3.5
(HL)	2	7	3.5
(IX + d)	5	19	9.5
(IY + d)	5	19	9.5

Adressierungsart: r: implizit; n: unmittelbar; (HL): indirekt; (IX + d), (IY + d): indiziert.

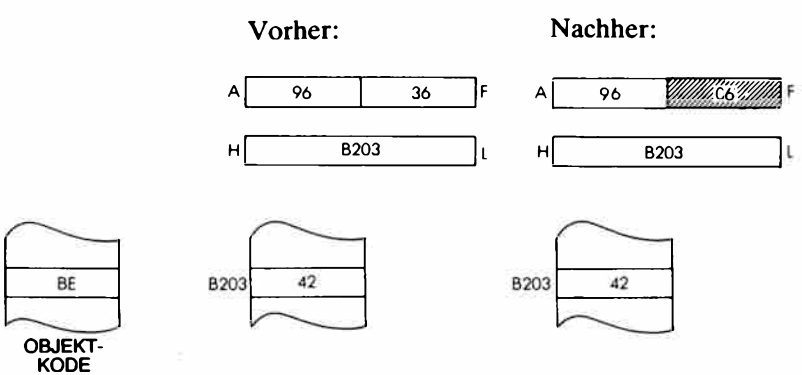
Byte Kode: CP r:

r:	A	B	C	D	E	H	L
	BF	B8	B9	BA	BB	BC	BD

Flags:

S	Z		H		P/V	N	C
●	●		●		●	1	●

Beispiel: CP (HL)



CPD

Vergleiche und dekrementiere.

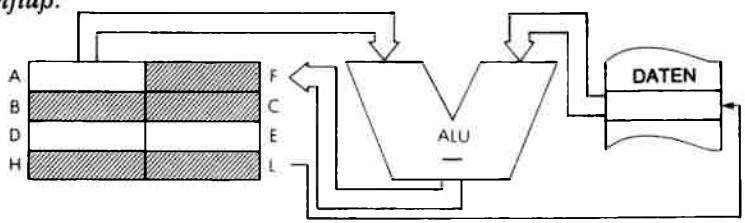
Funktion: A — [HL]; HL ← HL — 1; BC ← BC — 1

Format:

1	1	1	0	1	1	0	1	Byte 1: ED
1	0	1	0	1	0	0	1	Byte 2: A9

Beschreibung: Der Inhalt der Speicherstelle, die durch den Inhalt des Registerpaares HL adressiert wird, wird vom Inhalt des Akkumulators subtrahiert, das Ergebnis wird nicht weiter berücksichtigt. Danach werden die Registerpaare HL und BC dekrementiert.

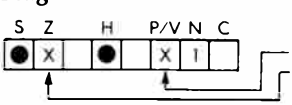
Datenfluß:



Befehlsablauf: 4 M Zyklen; 16 T Zustände: 8 µsek @ 2 MHz

Adressierungsart: Indirekt.

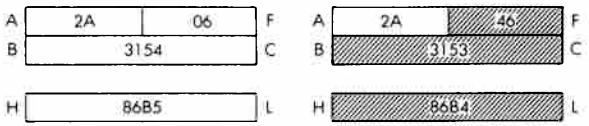
Flags:

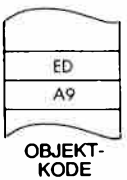


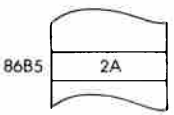
Zurückgesetzt, wenn BC = 0 nach der Ausführung, sonst gesetzt
Gesetzt, wenn A = [HL]

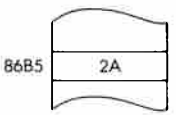
Beispiel: CPD

Vorher: **Nachher:**









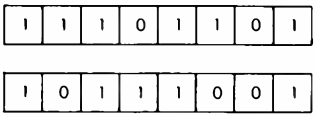
CPDR

Blockvergleich und Dekrementieren.

Funktion:

A – [HL]; HL←HL – 1; BC←BC – 1;
Wiederhole bis BC = 0 oder A = [HL]

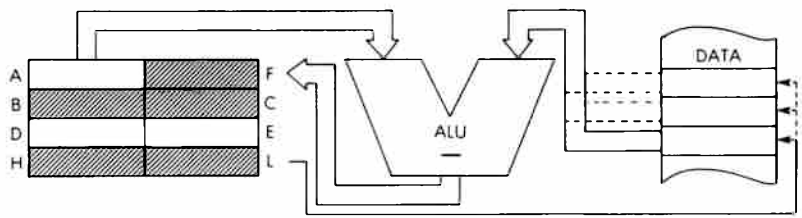
Format:



Beschreibung:

Der Inhalt der Speicherstelle, die durch das Registerpaar HL adressiert wird, wird vom Inhalt des Akkumulators subtrahiert, das Ergebnis wird nicht weiter berücksichtigt. Danach werden die Registerpaare BC und HL dekrementiert ist. BC ≠ 0 und A ≠ [HL], dann wird der Befehlszähler um zwei dekrementiert und der Befehl nochmals ausgeführt.

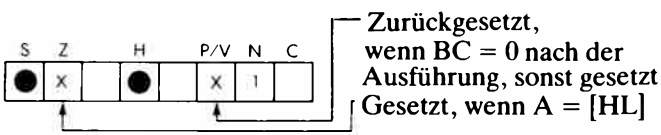
Datenfluß:



Befehlsablauf:

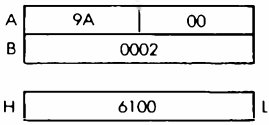
BC = 0 oder A = [HL]:4 M Zyklen; 16 T Zustände: 8 µsek @ 2 MHz
BC ≠ 0 und A ≠ [HL]:5 M Zyklen; 21 T Zustände: 10,5 µsek @ 2 MHz

Flags:

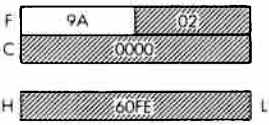


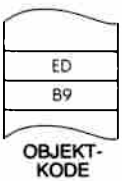
Beispiel: CPDR

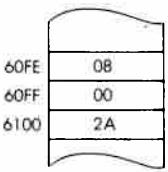
Vorher:

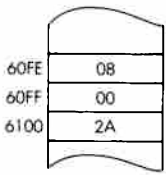


Nachher:









CPI

Vergleiche und inkrementiere.

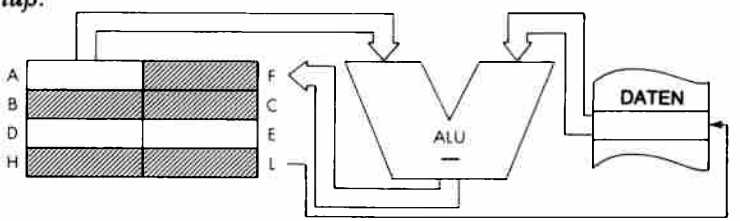
Funktion: A — [HL]; HL ← HL + 1; BC ← BC — 1

Format:

1	1	1	0	1	1	0	1	*Byte 1: ED*
1	0	1	0	0	0	0	1	*Byte 2: A1*

Beschreibung: Der Inhalt der Speicherzelle, die durch das Registerpaar HL adressiert wird, wird vom Inhalt des Akkumulators subtrahiert, das Ergebnis wird nicht weiter berücksichtigt. Das Registerpaar HL wird inkrementiert und das Registerpaar BC dekrementiert.

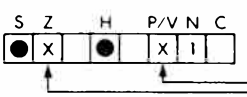
Datenfluß:



Befehlsablauf: 4 M Zyklen; 16 T Zustände: 8 µsek @ 2 MHz

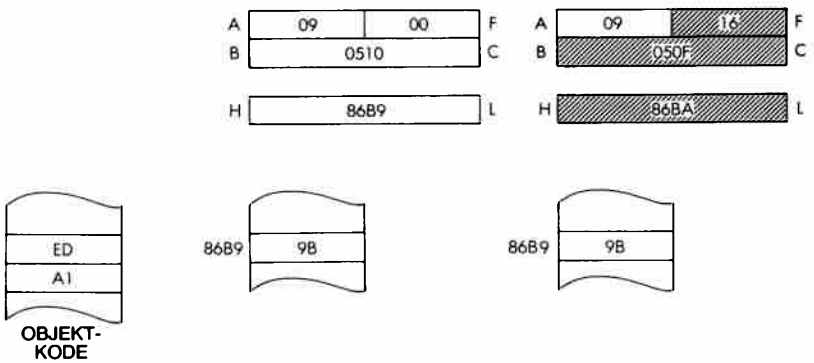
Adressierungsart: Indirekt.

Flags:



Zurückgesetzt, wenn BC = 0 nach der Ausführung, sonst gesetzt
Gesetzt, wenn A = [HL]

Beispiel: CPI



CPIR

Blockvergleich und Inkrementieren.

Funktion:

A − [HL]; HL←HL + 1; BC←BC − 1;
Wiederhole bis BC = 0 oder A = [HL]

1	1	1	0	1	1	0	1

Byte 1: ED

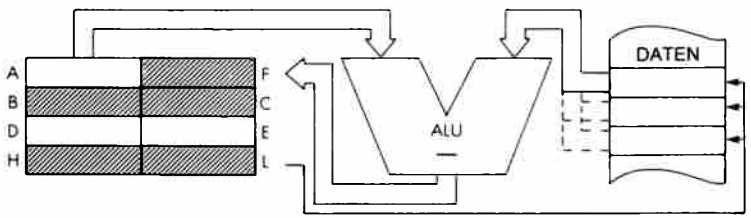
1	0	1	1	0	0	0	1

Byte 2: B1

Beschreibung:

Der Inhalt der Speicherstelle, die durch das Registerpaar HL adressiert wird, wird vom Inhalt des Akkumulators subtrahiert, das Ergebnis wird nicht weiter berücksichtigt. Danach wird das Registerpaar HL inkrementiert und das Registerpaar BC dekrementiert. Ist BC ≠ 0 und A ≠ [HL], dann wird der Befehlszähler um zwei dekrementiert und der Befehl nochmals ausgeführt.

Datenfluß:



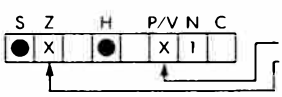
Befehlsablauf:

BC = 0 oder A = [HL]:4 M Zyklen; 16 T Zustände: 8 µsek @ 2 MHz
BC ≠ 0 und A ≠ [HL]:5 M Zyklen; 21 T Zustände: 10,5 µsek @ 2 MHz

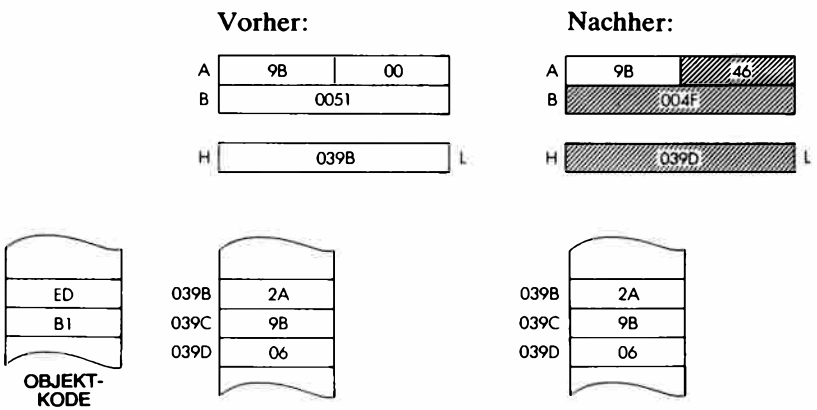
Adressierungsart:

Indirekt.

Flags:



Beispiel: CPIR



CPL

Komplementiere Akkumulator.

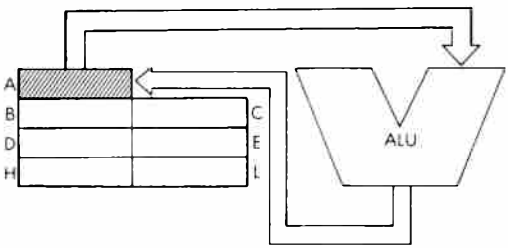
Funktion: $A \leftarrow \overline{A}$

Format:

0	0	1	0	1	1	1	1	2F

Beschreibung: Der Inhalt des Akkumulators wird komplementiert oder invertiert und das Ergebnis wieder im Akkumulator gespeichert (Einerkomplement).

Datenfluß:



Befehlsablauf: 1 M Zyklen; 4 T Zustände; 2 μsek @ 2 MHz

Adressierungsart: Implizit.

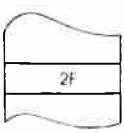
Flags:

S	Z		H		P/V	N	C
			1			1	

Beispiel: CPL

Vorher: A 3D

Nachher: A C2



DAA

Dezimalanpassung des Akkumulators.

Funktion: Siehe unten.

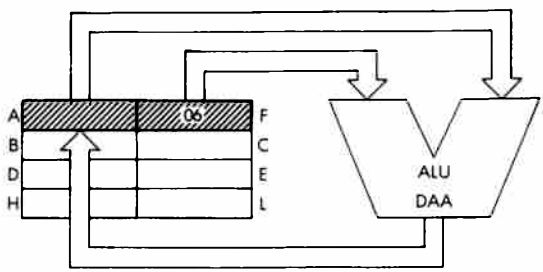
Format:

0	0	1	0	0	1	1	1	27

Beschreibung: Dieser Befehl addiert bedingt „6" zum rechten und/oder linken Nibble des Akkumulators, abhängig vom Inhalt des Statusregisters. Dieser Befehl dient zur BCD-Umwandlung nach arithmetischen Operationen.

N	C	Wert des oberen Nibble	H	Wert des unteren Nibble	# zu A addiert	C nach Ausführung
0	0	0-9	0	0-9	00	0
(ADD,	0	0-8	0	A-F	06	0
ADC,	0	0-9	1	0-3	06	0
INC)	0	A-F	0	0-9	60	1
	0	9-F	0	A-F	66	1
	0	A-F	1	0-3	66	1
	1	0-2	0	0-9	60	1
	1	0-2	0	A-F	66	1
	1	0-3	1	0-3	66	1
1	0	0-9	0	0-9	00	0
(SUB,	0	0-8	1	6-F	FA	0
SBC,	1	7-F	0	0-9	AO	1
DEC,	1	6-F	1	6-F	9A	1
NEG)						

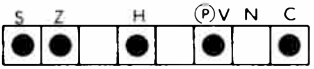
Datenfluß:



Befehlsablauf: 1 M Zyklen; 4 T Zustände; 2 μsek @ 2 MHz

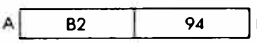
Adressierungsart: Implizit.

Flags:



Beispiel: DAA

Vorher:



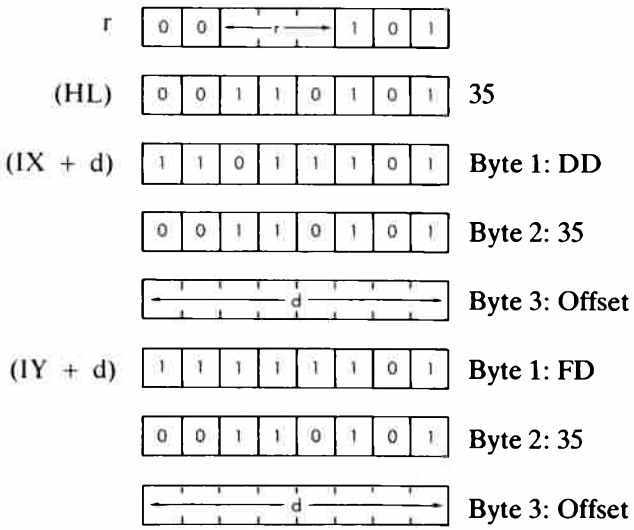
Nachher:

DEC m

Dekrementiere Operand m.

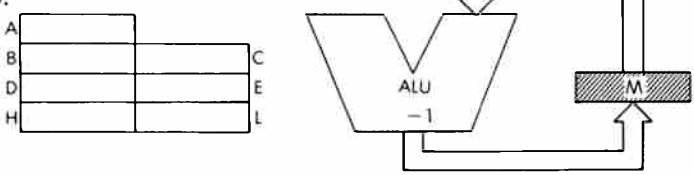
Funktion: m ← m − 1

Format: m: kann sein r,(HL),(IX + d),(IY + d)



Beschreibung: Der Inhalt der Speicherstelle, die durch den angegebenen Operanden adressiert wird, wird dekrementiert und wieder an dieser Stelle gespeichert. m ist bei der Beschreibung des Befehls INC definiert.

Datenfluß:



Befehlsablauf:

m:	M Zyklen:	T Zustände:	µsek @ 2 MHz:
r	1	4	2
(HL)	3	11	5.5
(IX + d)	6	23	11.5
(IY + d)	6	23	11.5

Adressierungsart: r: implizit; (HL): indirekt; (IX + d), (IY + d): indiziert.

Byte Kode: DEC r

r:	A	B	C	D	E	H	L
	3D	05	0D	15	1D	25	2D

Flags:

S	Z		H		P/V	N	C
●	●		●		●	1	

Beispiel: DEC C

Vorher:

0F	C

Nachher:

0E	C

DEC rr

Dekrementiere Registerpaar rr.

Funktion:

rr ← rr − 1

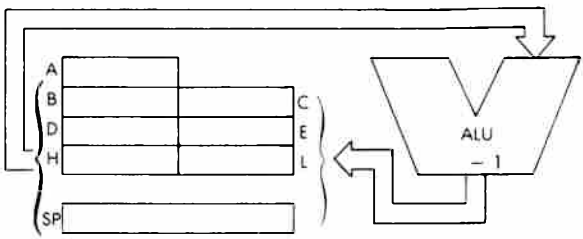
Format:

0	0	r	r	1	0	1	1

Beschreibung: Der Inhalt des angegebenen Registerpaares wird dekrementiert und das Ergebnis wieder in dem Registerpaar gespeichert. rr kann sein:

BC − 00	HL − 10
DE − 01	SP − 11

Datenfluß:



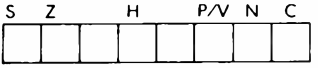
Befehlsablauf: 1 M Zyklus; 6 T Zustände; 3 μsek @ 2 MHz

Adressierungsart: Implizit.

Byte Kode: rr :

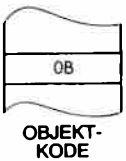
BC	DE	HL	SP
0B	1B	2B	3B

Flags:

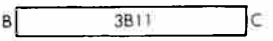


(kein Einfluß)

Beispiel: DEC BC



Vorher:



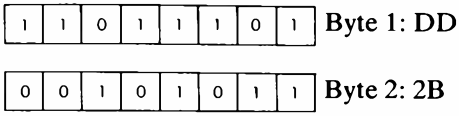
Nachher:

DEC IX

Dekrementiere IX.

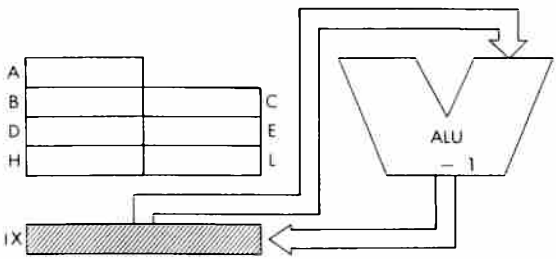
Funktion: IX ← IX − 1

Format:



Beschreibung: Der Inhalt des Registers IX wird dekrementiert und das Ergebnis wieder im Register IX gespeichert.

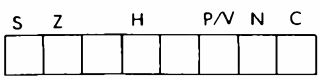
Datenfluß:



Befehlsablauf: 2 M Zyklen; 10 T Zustände; 5 µsek @ 2 MHz

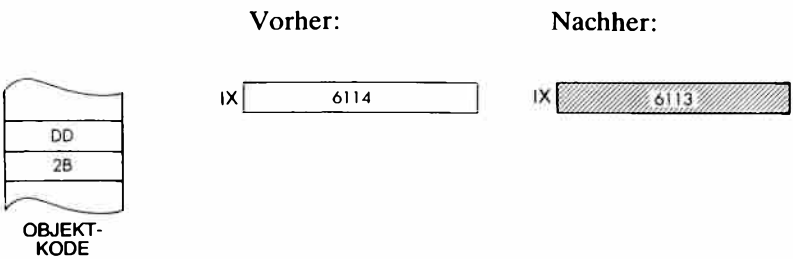
Adressierungsart: Implizit.

Flags:



(kein Einfluß)

Beispiel: DEC IX

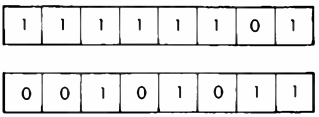


DEC IY

Dekrementiere IY.

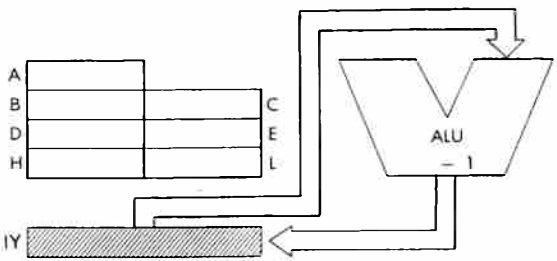
Funktion: IY ← IY − 1

Format:



Beschreibung: Der Inhalt des Registers IY wird dekrementiert und das Ergebnis wieder im Register IY gespeichert.

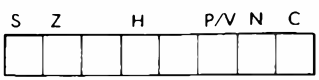
Datenfluß:



Befehlsablauf: 2 M Zyklen; 10 T Zustände; 5 µsek @ 2 MHz

Adressierungsart: Implizit.

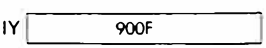
Flags:



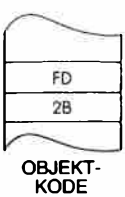
(kein Einfluß)

Beispiel: DEC IY

Vorher:



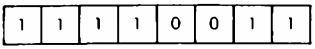
Nachher:



DI Sperre Interrupt.

Funktion: IFF ← 0

Format:



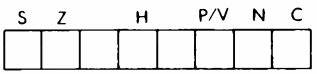
F3

Beschreibung: Die Interrupt-Flip-Flops werden zurückgesetzt und damit alle maskierbaren Interrupts gesperrt. Nach Ausführung dieses Befehls ist der maskierbare Interrupt gesperrt, bis er durch einen Befehl EI wieder freigegeben wird.

Befehlsablauf: 1 M Zyklen; 4 T Zustände; 2 μsek @ 2 MHz

Adressierungsart: Implizit.

Flags:

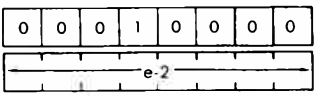


(kein Einfluß)

DJNZ e Dekrementiere B und springe falls nicht Null.

Funktion: B ← B − 1; falls B ≠ 0: PC ← PC + e

Format:

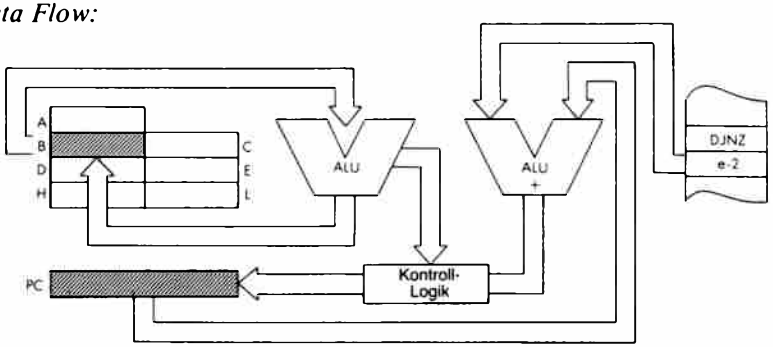


Byte 1: 10

Byte 2: Offset

Beschreibung: Das Register B wird dekrementiert. Ist das Ergebnis nicht Null, dann wird der Offset mit Zweierkomplement-Arithmetik zum Befehlszähler addiert, so daß sowohl vorwärts als auch rückwärts gesprungen werden kann. Der Offset wird addiert zu PC + 2 (nach dem Sprung). Deshalb liegt der effektive Offset zwischen −126 und +129 Bytes. Der Assembler subtrahiert automatisch vom Quell-Offset, um den Hex-Kode zu erzeugen.

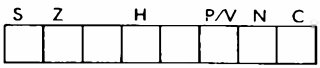
Data Flow:



Befehlsablauf: B ≠ 0: 3 M Zyklen; 13 T Zustände; 6,5 µsek @ 2 MHz
B = 0: 2 M Zyklen; 8 T Zustände; 4 µsek @2 MHz

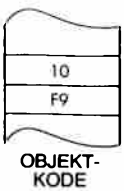
Adressierungsart: Unmittelbar.

Flags:

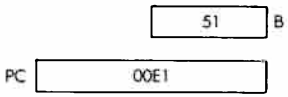


(kein Einfluß)

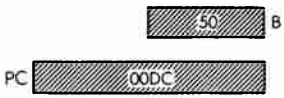
Beispiel: DJNZ $ - 5 ($ = current PC)



Vorher:



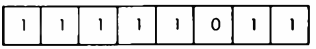
Nachher:



EI Gib Interrupts frei.

Funktion: IFF ← 1

Format:



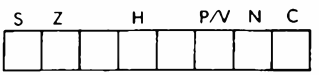
FB

Beschreibung: Die Interrupt-Flip-Flops werden gesetzt und dadurch nach der Ausführung des Befehls, der auf EI folgt, maskierbare Interrupts freigegeben. Bis zu diesem Zeitpunkt sind maskierbare Interrupts gesperrt.

Befehlsablauf: 1 M Zyklen; 4 T Zustände; 2 μsek @ 2 MHz

Adressierungsart: Implizit.

Flags:



(kein Einfluß)

Beispiel: *Eine übliche Folge am Ende einer Interrupt-Routine ist:*
EI
RETI
Nach Ausführung von RETI sind maskierbare Interrupts wieder freigegeben.

EX AF, AF' Vertausche Akkumulator und Flags mit den Zweitregistern.

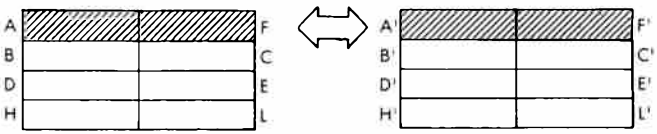
Funktion: AF↔AF'

Format:

0	0	0	0	1	0	0	0	08

Beschreibung: Der Inhalt von Akkumulator und Statusregister wird mit dem dem Inhalt der entsprechenden Zweitregister vertauscht.

Datenfluß:



Befehlsablauf: 1 M Zyklen; 4 T Zustände; 2 μsek @ 2 MHz

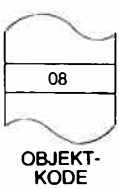
Adressierungsart: Implizit.

Flags:

S	Z		H		P/V	N	C

(kein Einfluß)

Beispiel: EX AF, AF'



Vorher:

A	04	81	F
A'	90	3A	F'

Nachher:

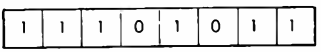
A	90	3A	F
A'	04	81	F'

EX DE, HL

Vertausche Register HL und DE.

Funktion: DE ⟷ HL

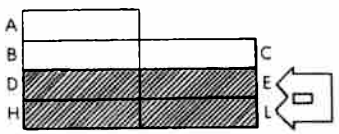
Format:



EB

Beschreibung: Der Inhalt von der Registerpaare HL und DE wird vertauscht.

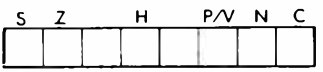
Datenfluß:



Befehlsablauf: 1 M Zyklen; 4 T Zustände; 2 µsek @ 2 MHz

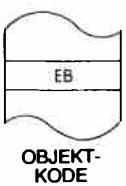
Adressierungsart: Implizit.

Flags:

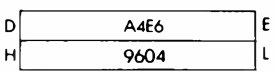


(kein Einfluß)

Beispiel: EX DE, HL



Vorher:



Nachher:

EX (SP), HL Vertausche HL mit dem Inhalt des obersten Stapelelementes.

Funktion: (SP) ↔ L; (SP + 1) ↔ H

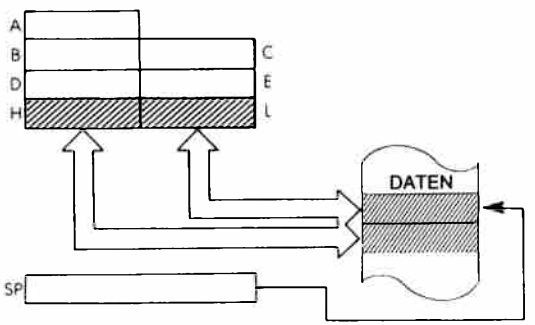
Format:

1	1	1	0	0	0	1	1

E3

Beschreibung: Der Inhalt des Registers L wird mit dem Inhalt der Speicherstelle vertauscht, die der Stapelzeiger adressiert. Der Inhalt des Registers H wird mit dem Inhalt der Speicherstelle vertauscht, die auf die durch den Stapelzeiger adressierte folgt.

Datenfluß:



Befehlsablauf: 5 M Zyklen; 19 T Zustände; 9,5 µsek @ 2 MHz

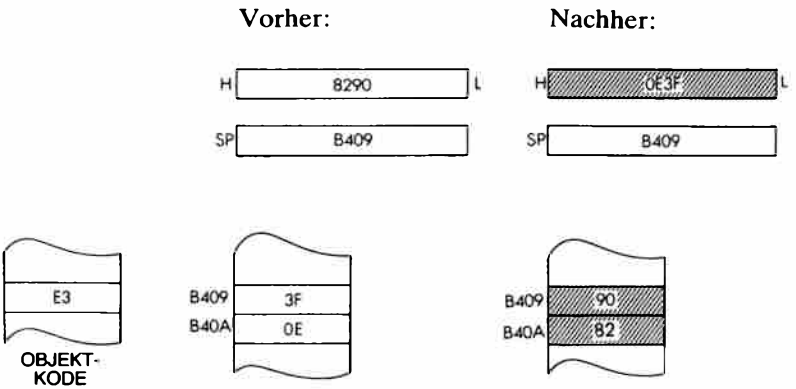
Adressierungsart: Indirekt.

Flags:

S	Z		H		P/V	N	C

(kein Einfluß)

Beispiel: EX (SP), HL



EX (SP), IX

Vertausche IX mit dem Inhalt des obersten Stapelelementes.

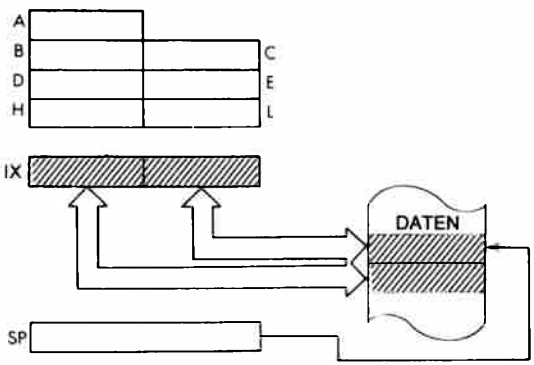
Funktion: $(SP) \leftrightarrow IX_{unten}; (SP + 1) \leftrightarrow IX_{oben}$

Format:

1	1	0	1	1	1	0	1	Byte 1: DD
1	1	1	0	0	0	1	1	Byte 2: E3

Beschreibung: Der Inhalt der unteren Hälfte des Registers IX wird mit dem Inhalt der Speicherzelle vertauscht, die durch den Stapelzeiger adressiert wird. Der Inhalt der oberen Hälfte des Registers IX wird mit dem Inhalt der Speicherstelle vertauscht, die unmittelbar auf die durch den Stapelzeiger adressierte folgt.

Datenfluß:



Befehlsablauf: 6 M Zyklen; 23 T Zustände; 11,5 µsek @ 2 MHz

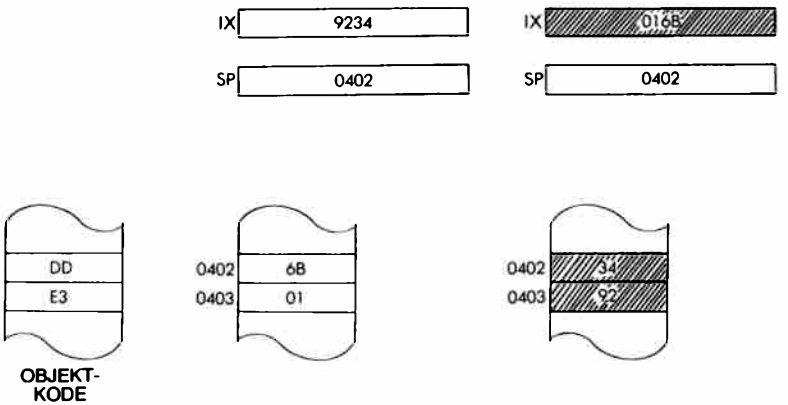
Adressierungsart: Indirekt.

Flags:

S	Z		H		P/V	N	C

(kein Einfluß)

Beispiel: EX (SP), IX



EX (SP), IY Vertausche IY mit dem Inhalt des obersten Stapelelementes.

Funktion: $(SP) \leftrightarrow IY_{unten}; (SP + 1) \leftrightarrow IY_{oben}$

Format:

1	1	1	1	1	1	0	1

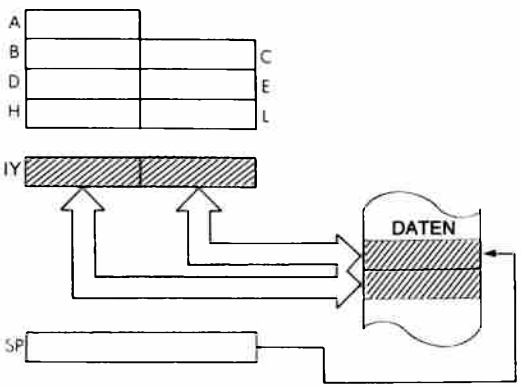
Byte 1: FD

1	1	1	0	0	0	1	1

Byte 2: E3

Beschreibung: Der Inhalt der unteren Hälfte des Registers IY wird mit dem Inhalt der Speicherzelle vertauscht, die durch den Stapelzeiger adressiert wird. Der Inhalt der oberen Hälfte des Registers IY wird mit dem Inhalt der Speicherstelle vertauscht, die unmittelbar auf die durch den Stapelzeiger adressierte folgt.

Datenfluß:



Befehlsablauf: 6 M Zyklen; 23 T Zustände; 11,5 µsek @ 2 MHz

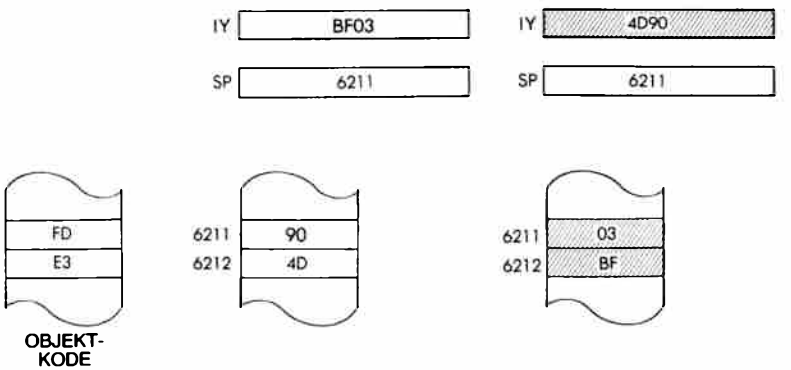
Adressierungsart: Indirekt.

Flags:

S	Z		H		P/V	N	C

(kein Einfluß)

Beispiel: EX (SP), IY



EXX

Vertausche mit den Zweitregistern.

Funktion: BC ↔ BC'; DE ↔ DE'; HL ↔ HL'

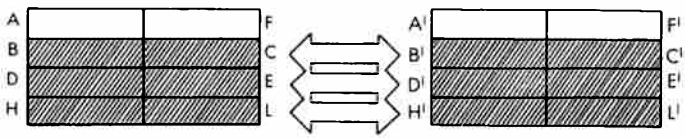
Format:

1	1	0	1	1	0	0	1

D9

Beschreibung: Der Inhalt der Universalregister wird mit dem Inhalt der entsprechenden Zweitregister vertauscht.

Datenfluß:



Befehlsablauf: 1 M Zyklen; 4 T Zustände; 2 μsek @ 2 MHz

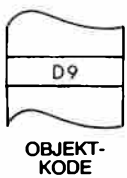
Adressierungsart: Implizit.

Flags:

S	Z		H		P/V	N	C

(kein Einfluß)

Beispiel: EXX



Vorher:

A	04	2B	F
B	39	26	C
D	54	02	E
H	F1	D0	L

A'	3F	2A	F'
B'	8C	00	C'
D'	93	D0	E'
H'	4F	E3	L'

Nachher:

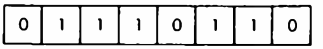
A	04	2B	F
B	8C	00	C
D	93	D0	E
H	4F	E3	L

A'	3F	2A	F'
B'	39	26	C'
D'	54	02	E'
H'	F1	D0	L'

HALT Halte CPU an.

Funktion: CPU angehalten.

Format:



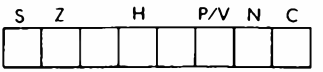
76

Beschreibung: Die CPU unterbricht die Operation und führt solange NOPs aus, bis ein Interrupt oder ein Reset eintritt, um weiterhin Memory Refresh Zyklen zu liefern.

Befehlsablauf: 1 M Zyklen; 4 T Zustände; 2 µsek @ 2 MHz + undefiniert NOPs.

Adressierungsart: Implizit.

Flags:

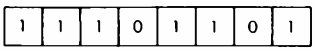


(kein Einfluß)

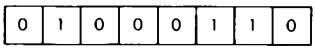
IM 0 Setze Interrupt Modus 0.

Funktion: Interne Interruptkontrolle.

Format:



Byte 1: ED



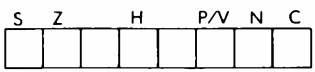
Byte 2: 46

Beschreibung: Setzt Interrupt Modus 0. In dieser Betriebsart kann der unterbrechende Baustein einen Befehl zur Ausführung auf den Datenbus legen. Das erste Byte des Befehls muß während des Interrupt Acknowledge Zyklus erscheinen.

Befehlsablauf: 2 M Zyklen; 8 T Zustände; 4 µsek @2 MHz

Adressierungsart: Implizit.

Flags:



(kein Einfluß)

IM 1 Setze Interrupt Modus 1.

Funktion: Interne Interruptkontrolle.

Format:

1	1	1	0	1	1	0	1

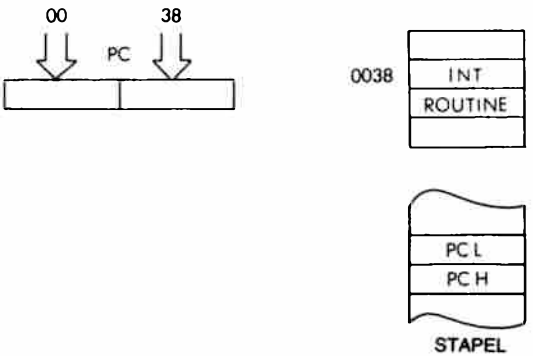
Byte 1: ED

0	1	0	1	0	1	1	0

Byte 2: 56

Beschreibung: Setzt Interrupt Modus 1. Wenn ein Interrupt auftritt, wird ein Befehl RST 0038H ausgeführt.

Datenfluß:



Befehlsablauf: 2 M Zyklen; 8 T Zustände; 4 μsek @2 MHz

Adressierungsart: Implizit.

Flags:

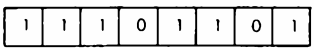
S	Z		H		P/V	N	C

(kein Einfluß)

IM 2 Setze Interrupt Modus 2.

Funktion: Interne Interruptkontrolle.

Format:



Byte 1: ED

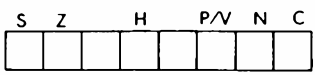
Byte 2: 5E

Beschreibung: Setzt Interrupt Modus 2. Wenn ein Interrupt auftritt, dann muß der unterbrechende Baustein ein Datenbyte liefern, das als untere Hälte einer Adresse verwendet wird. Die obere Hälfte dieses Adreßvektors wird aus dem Register I entnommen. Dieser zeigt auf eine weitere Adresse, die im Speicher abgelegt ist. Diese Adresse wird in den Befehlszähler geladen und dort beginnt die Ausführung.

Befehlsablauf: 2 M Zyklen; 8 T Zustände; 4 μsek @2 MHz

Adressierungsart: Implizit.

Flags:



(kein Einfluß)

IN r, (C)

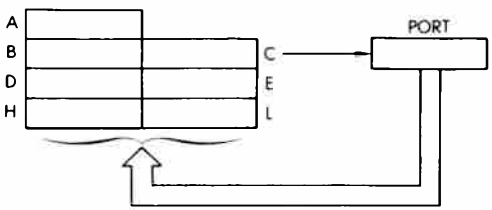
Lade Register r aus Port (C).

Funktion: r ← (C)

Format:

Beschreibung: Das periphere Gerät, das durch den Inhalt des Registers C adressiert wird, wird gelesen und das Ergebnis in das angegebene Register geladen.
C liefert die Bits A0 bis A7 des Adreßbusses,
B liefert Bit A8 bis A15.

Datenfluß:



r kann sein:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

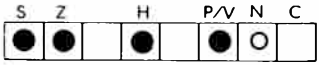
Befehlsablauf: 3 M Zyklen; 12 T Zustände; 6 µsek @2 MHz

Adressierungsart: Extern.

Byte Kode:

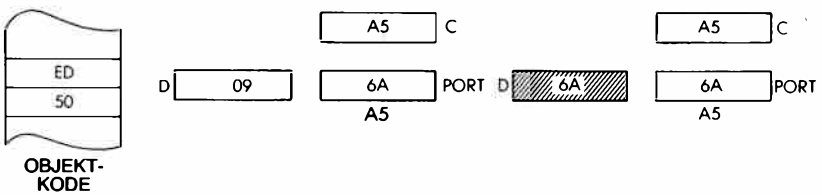
r:	A	B	C	D	E	H	L
ED	78	40	48	50	58	60	68

Flags:



Es ist wichtig zu beachten, daß IN A,(N) keinerlei Einfluß auf die Flags hat, dagegen aber IN r,(C).

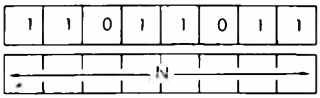
Beispiel: IN D, (C)



IN A, (N) Lade Akkumulator aus Port N.

Funktion: A ← (N)

Format:

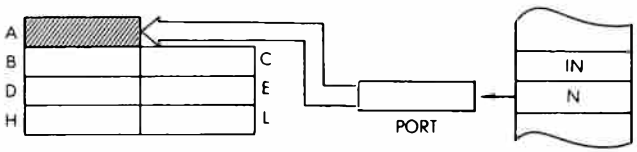


Byte 1: DB

Byte 2: Portadresse:

Beschreibung: Das periphere Gerät N wird gelesen und das Ergebnis in den Akkumulator geladen. Das Literal N liegt auf Bit A0 bis A7 des Adreßbusses, A liefert Bit A8 bis A15.

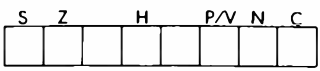
Datenfluß:



Befehlsablauf: 3 M Zyklen; 11 T Zustände; 5,5 µsek @2 MHz

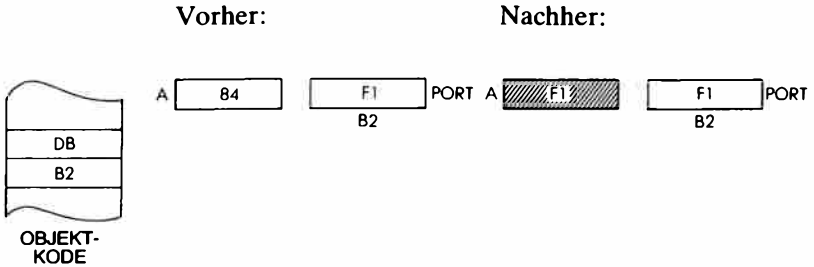
Adressierungsart: Extern.

Flags:



(kein Einfluß)

Beispiel: IN A, (B2)

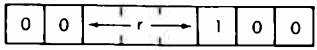


INC r

Inkrementiere Register r.

Funktion: r ← r + 1

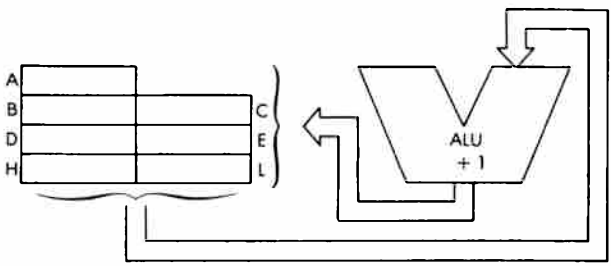
Format:



Beschreibung: Der Inhalt des angegebenen Registers wird inkrementiert. r kann sein:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

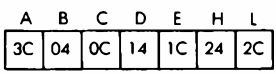
Datenfluß:



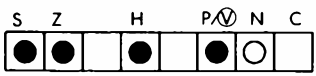
Befehlsablauf: 1 M Zyklus; 4 T Zustände; 2 μsek @2 MHz

Adressierungsart: Implizit.

Byte Kode:



Flags:



Beispiel: INC D

Vorher: D 06

Nachher: D 07

INC rr

Inkrementiere Registerpaar rr.

Funktion: rr ← rr + 1

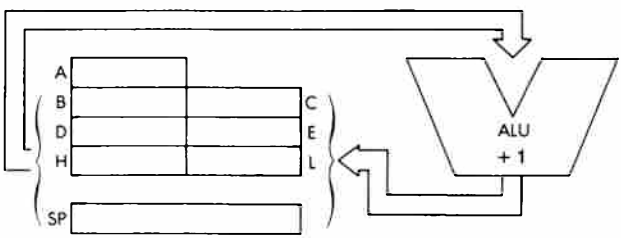
Format:

0	0	r	r	0	0	1	1

Beschreibung: Der Inhalt des angegebenen Registerpaars wird inkrementiert und das Ergebnis wieder in dem Registerpaar gespeichert. r kann sein:

BC – 00	HL – 10
DE – 01	SP – 11

Datenfluß:



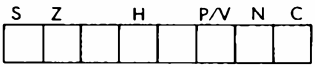
Befehlsablauf: 1 M Zyklus; 6 T Zustände; 3 µsek @2 MHz

Adressierungsart: Implizit.

Byte Kode: rr:

BC	DE	HL	SP
03	13	23	33

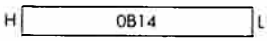
Flags:



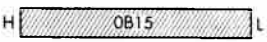
(kein Einfluß)

Beispiel: INC HL

Vorher:



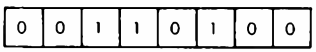
Nachher:



INC (HL) Inkrementiere die indirekt adressierte Speicherstelle (HL).

Funktion: (HL) ← (HL) + 1

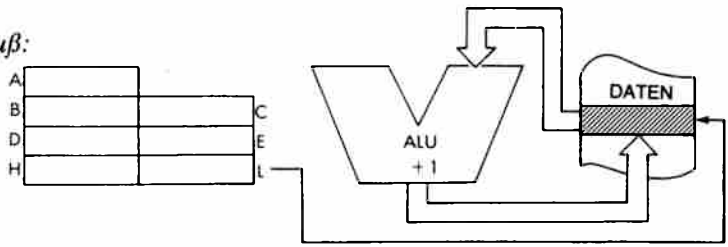
Format:



34

Beschreibung: Der Inhalt der Speicherstelle, die durch das Registerpaar HL adressiert wird, wird inkrementiert und das Ergebnis wieder in dieser Stelle gespeichert.

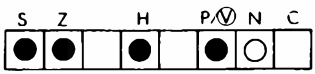
Datenfluß:



Befehlsablauf: 3 M Zyklen; 11 T Zustände; 5,5 µsek @2 MHz

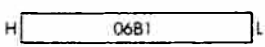
Adressierungsart: Indirekt.

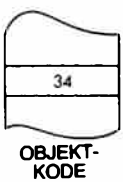
Flags:

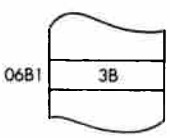


Beispiel: INC (HL)

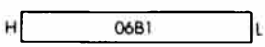
Vorher:

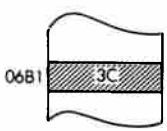






Nachher:

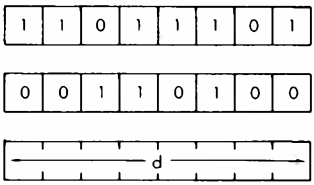




INC (IX + d) Inkrementiere die indiziert adressierte Speicherstelle (IX + d).

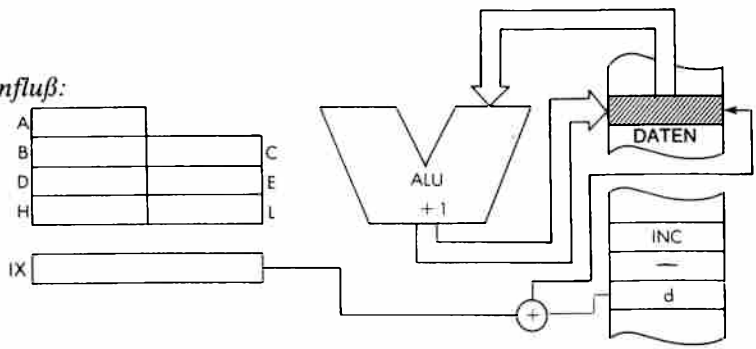
Funktion: (IX + d) ← (IX + d) + 1

Format:



Beschreibung: Der Inhalt der Speicherstelle, die durch den Inhalt des Registers IX plus dem angegebenen Offset adressiert wird, wird inkrementiert und das Ergebnis wieder in dieser Stelle gespeichert.

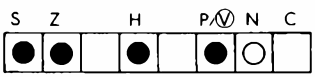
Datenfluß:



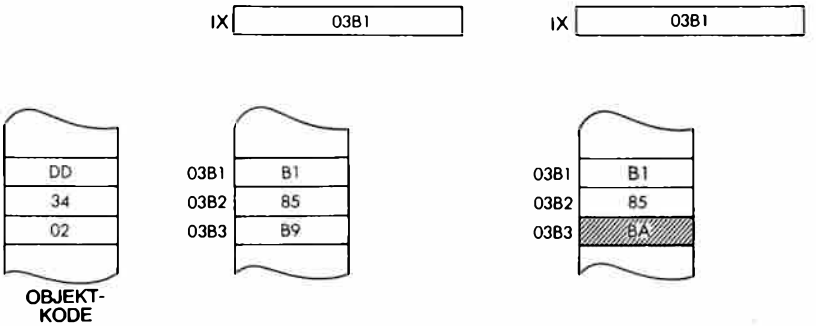
Befehlsablauf: 6 M Zyklen; 23 T Zustände; 11,5 µsek @2 MHz

Adressierungsart: Indiziert.

Flags:



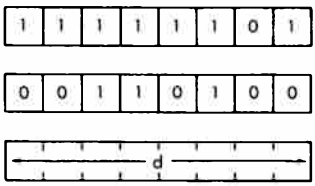
Beispiel: INC (IX + 2)



INC (IY + d) Inkrementiere die indiziert adressierte Speicherstelle (IY + d).

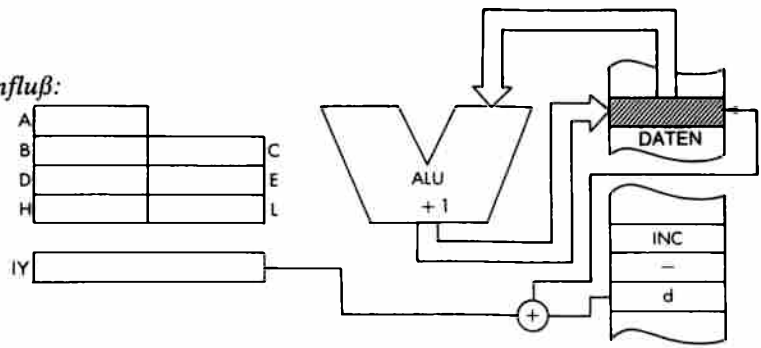
Funktion: (IY + d) ← (IY + d) + 1

Format:



Beschreibung: Der Inhalt der Speicherstelle, die durch den Inhalt des Registers IY plus dem angegebenen Offset adressiert wird, wird inkrementiert und das Ergebnis wieder in dieser Stelle gespeichert.

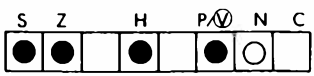
Datenfluß:



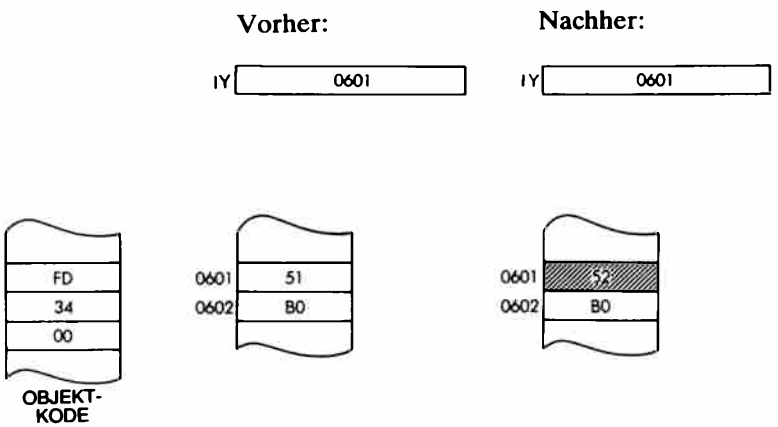
Befehlsablauf: 6 M Zyklen; 23 T Zustände; 11,5 µsek @2 MHz

Adressierungsart: Indiziert.

Flags:



Beispiel: INC (IY + 0)

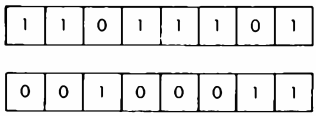


INC IX

Inkrementiere IX.

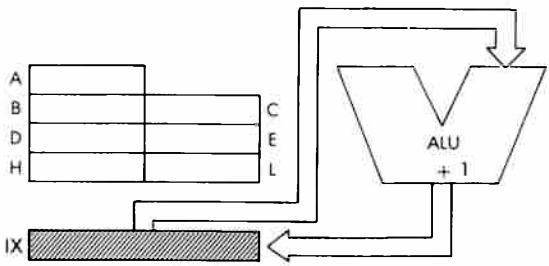
Funktion: IX ← IX + 1

Format:



Beschreibung: Der Inhalt des Registers IX wird inkrementiert und das Ergebnis wieder im Register IX gespeichert.

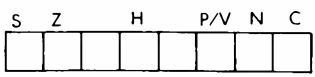
Datenfluß:



Befehlsablauf: 2 M Zyklen; 10 T Zustände; 5 µsek @2 MHz

Adressierungsart: Implizit.

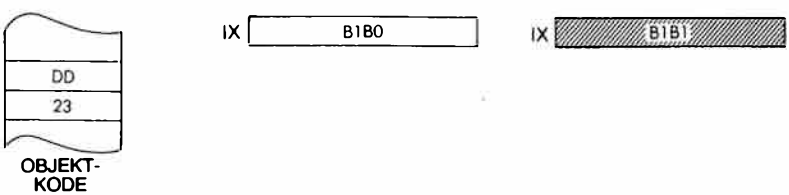
Flags:



(kein Einfluß)

Beispiel: INC IX

Vorher: Nachher:

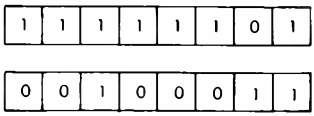


INC IY

Inkrementiere IY.

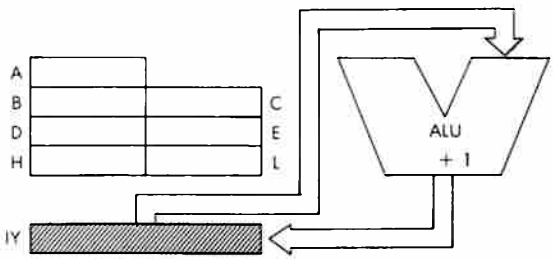
Funktion: IY ← IY + 1

Format:



Beschreibung: Der Inhalt des Registers IY wird inkrementiert und das Ergebnis wieder im Register IY gespeichert.

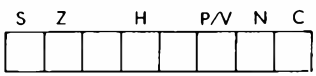
Datenfluß:



Befehlsablauf: 2 M Zyklen; 10 T Zustände; 5 µsek @2 MHz

Adressierungsart: Implizit.

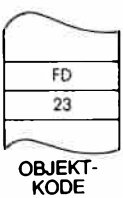
Flags:



(kein Einfluß)

Beispiel: INC IY

Vorher: Nachher:



IND

Eingabe mit Dekrementieren.

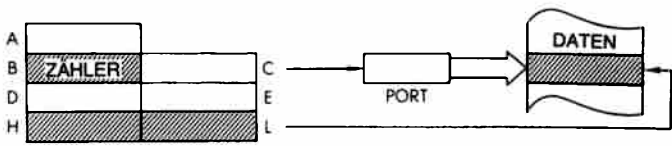
Funktion: (HL) ← (C); B ← B − 1; HL ← HL − 1

Format:

1	1	1	0	1	1	0	1	Byte 1: ED
1	0	1	0	1	0	1	0	Byte 2: AA

Beschreibung: Das periphere Gerät, das durch das Register C adressiert wird, wird gelesen und das Ergebnis in die Speicherstelle geladen, die durch das Registerpaar HL adressiert wird. Das Register B und das Registerpaar HL werden dann jeweils dekrementiert.

Datenfluß:



Befehlsablauf: 4 M Zyklen; 16 T Zustände; 8 µsek @2 MHz

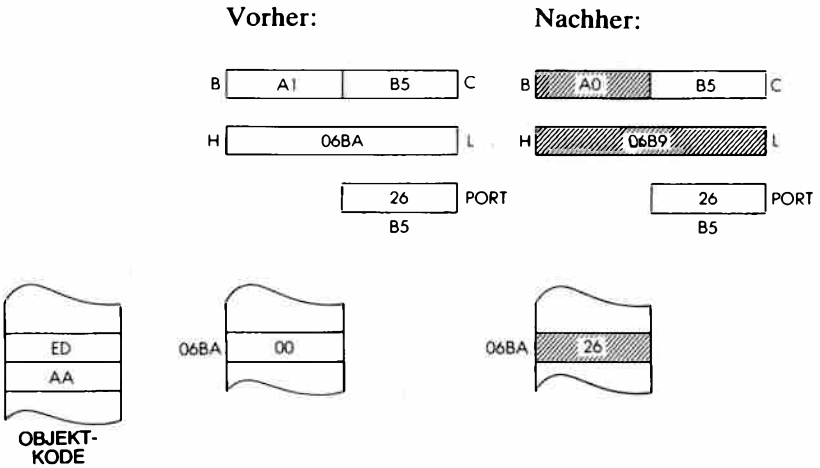
Adressierungsart: Extern.

Flags:

S	Z		H		P/V	N	C
?	X		?		?	1	

Gesetzt, wenn B=0 nach der Ausführung, sonst zurückgesetzt

Beispiel: IND



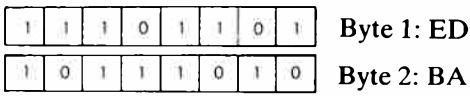
INDR

Blockeingabe mit Dekrementieren.

Funktion:

(HL) ← (C); B ← B − 1; HL ← HL − 1
Wiederholung, bis B = 0.

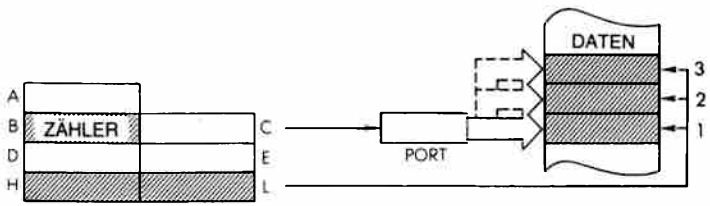
Format:



Beschreibung:

Das periphere Gerät, das durch das Register C adressiert wird, wird gelesen und das Ergebnis in die Speicherstelle geladen, die durch das Registerpaar HL adressiert wird. Das Register B und das Registerpaar HL werden dann dekrementiert. Ist B nicht Null, dann wird der Befehlszähler um zwei dekrementiert und der Befehl nochmals ausgeführt.

Datenfluß:



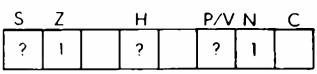
Befehlsablauf:

B = 0:4 M Zyklen; 16 T Zustände; 8 µsek @2 MHz
B ≠ 0:5 M Zyklen; 21 T Zustände; 10,5 µsek @2 MHz.

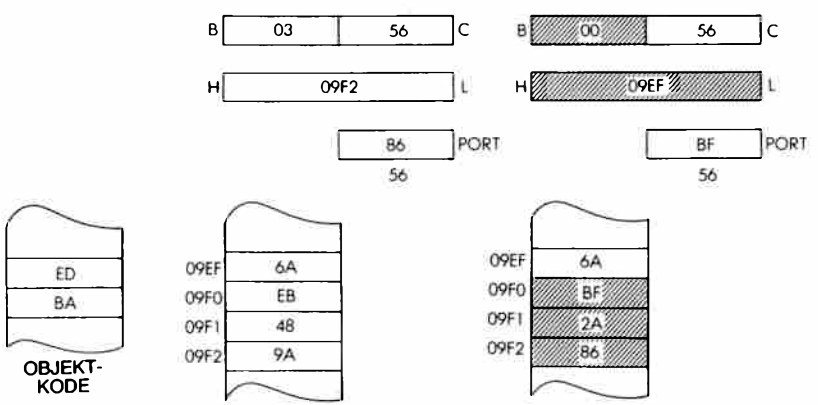
Adressierungsart:

Extern.

Flags:



Beispiel: INDR



INI Eingabe mit Inkrementieren.

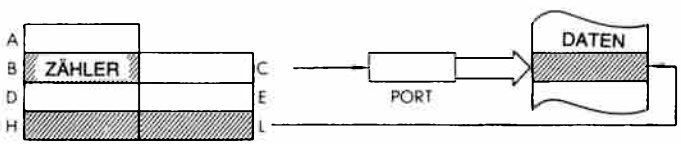
Funktion: (HL) ← (C); B ← B − 1; HL ← HL + 1

Format:

1	1	1	0	1	1	0	1	Byte 1: ED
1	0	1	0	0	0	1	0	Byte 2: A2

Beschreibung: Das periphere Gerät, das durch das Register C adressiert wird, wird gelesen und das Ergebnis in die Speicherstelle geladen, die durch das Registerpaar HL adressiert wird. Das Register B wird dann dekrementiert und das Registerpaar HL inkrementiert.
Der Inhalt von C wird auf die untere Hälfte des Adreßbusses gelegt, der Inhalt von B auf die obere Hälfte. Die I/O-Selektion wird generell mit C durchgeführt, d. h. mit A0 bis A7. B ist ein Bytezähler.

Datenfluß:



Befehlsablauf: 4 M Zyklen; 16 T Zustände; 8 μsek @2 MHz

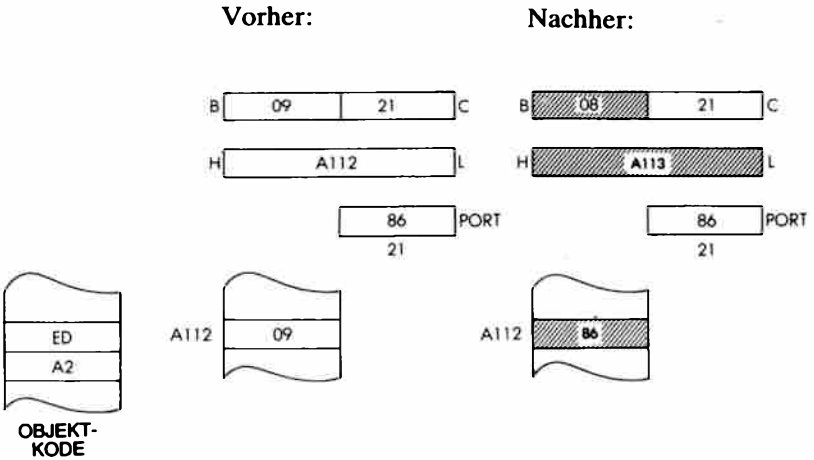
Adressierungsart: Extern.

Flags:

S	Z		H		P/V	N	C
?	X		?		?	1	

Z ist gesetzt, wenn B=0 nach der Ausführung, sonst zurückgesetzt.

Beispiel: INI



INIR

Blockeingabe mit Inkrementieren.

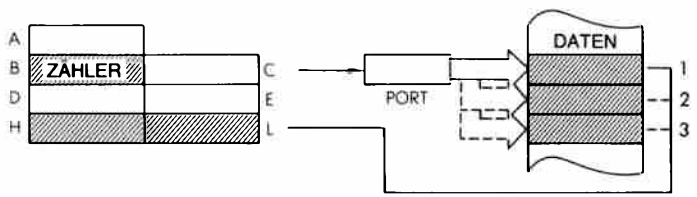
Funktion: (HL) ← (C); B ← B − 1; HL ← HL + 1; Wiederholung, bis B = 0.

Format:

1	1	1	0	1	1	0	1	Byte 1: ED
1	0	1	1	0	0	1	0	Byte 2: B2

Beschreibung: Das periphere Gerät, das durch das Register C adressiert wird, wird gelesen und das Ergebnis in die Speicherstelle geladen, die durch das Registerpaar HL adressiert wird. Das Register B wird dann dekrementiert und das Registerpaar HL inkrementiert. Ist B nicht Null, dann wird der Befehlszähler um zwei dekrementiert und der Befehl nochmals ausgeführt.

Datenfluß:



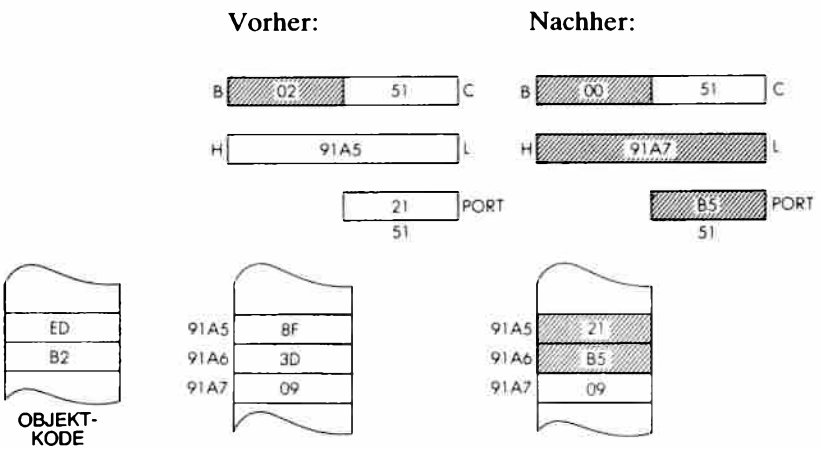
Befehlsablauf: B = 0:4 M Zyklen; 16 T Zustände; 8 µsek @2 MHz
B ≠ 0:5 M Zyklen; 21 T Zustände; 10,5 µsek @2 MHz

Adressierungsart: Extern.

Flags:

S	Z		H		P/V	N	C
?	1		?		?	1	

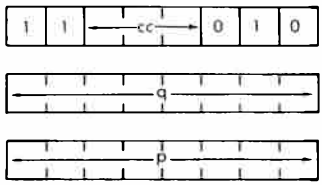
Beispiel: INIR



JP cc, pq Bedingter Sprung zur Adresse pq.

Funktion: Wenn cc erfüllt, dann PC←pq

Format:



Byte 1:

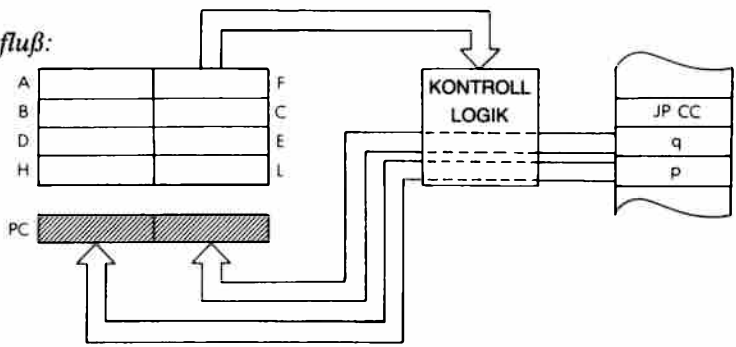
Byte 2: Adresse, untere Hälfte

Byte 3: Adresse, obere Hälfte

Beschreibung: Ist die angegebene Bedingung erfüllt, dann wird die Zweibyte-Adresse, die auf den Opcode folgt, in den Befehlszähler geladen. Dabei kommt das erste Byte hinter dem Opcode in die untere Hälfte von PC. Ist die Bedingung nicht erfüllt, dann wird die Adresse nicht beachtet. cc kann sein:

NZ – 000	nicht Null
Z – 001	Null
NC – 010	kein Übertrag (Carry)
C – 011	Übertrag (Carry)
PO – 100	ungerade Parität
PE – 101	gerade Parität
P – 110	plus
M – 111	minus

Datenfluß:



Befehlsablauf: 3 M Zyklen; 10 T Zustände; 5 µsek @2 MHz

Adressierungsart: Unmittelbar.

Byte Kode:

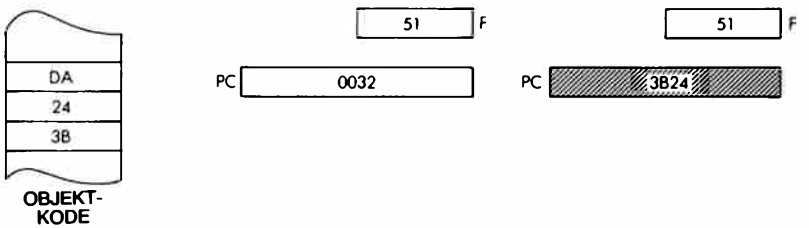
C C	NZ	Z	NC	C	PO	PE	P	M
	C2	CA	D2	DA	E2	EA	F2	FA

Flags:

S	Z		H		P/V	N	C

(kein Einfluß)

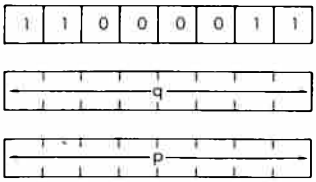
Beispiel: JP C, 3B24



JP pq Sprung zur Adresse pq.

Funktion: PC ← pq

Format:



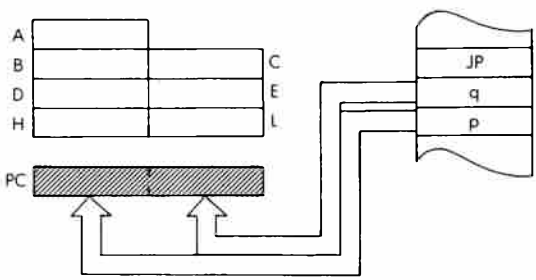
Byte 1: C3

Byte 2: Adresse, untere Hälfte

Byte 3: Adresse, obere Hälfte

Beschreibung: Der Inhalt der Speicherzelle unmittelbar hinter dem Opcode wird in die untere Hälfte, der Inhalt der Speicherzelle dahinter in die obere Hälfte des Befehlszählers geladen. Der nächste Befehl wird von dieser Adresse geholt.

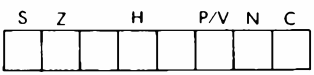
Datenfluß:



Befehlsablauf: 3 M Zyklen; 10 T Zustände; 5 µsek @2 MHz

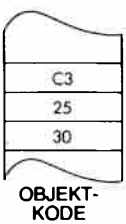
Adressierungsart: Unmittelbar.

Flags:

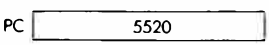


(kein Einfluß)

Beispiel: JP 3025



Vorher:

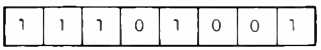


Nachher:

JP (HL) Sprung zur Adresse (HL)

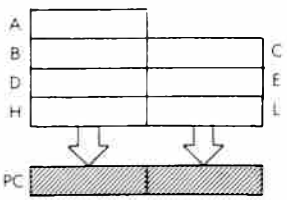
Funktion: PC ← HL

Format:



Beschreibung: Der Inhalt des Registerpaares HL wird in den Befehlszähler geladen. Der nächste Befehl wird von dieser Adresse geholt.

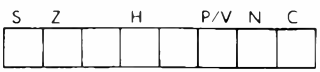
Datenfluß:



Befehlsablauf: 1 M Zyklus; 4 T Zustände; 2 µsek @2 MHz

Adressierungsart: Implizit.

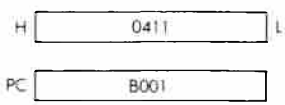
Flags:



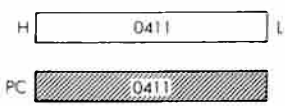
(kein Einfluß)

Beispiel: JP (HL)

Vorher:



Nachher:



JP (IX) Sprung zur Adresse (IX)

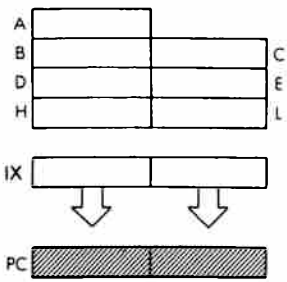
Funktion: PC ← IX

Format:

1	1	0	1	1	1	0	1	byte 1: DD
1	1	1	0	1	0	0	1	byte 2: E9

Beschreibung: Der Inhalt des Registers IX wird in den Befehlszähler geladen. Der nächste Befehl wird von dieser Adresse geholt.

Datenfluß:



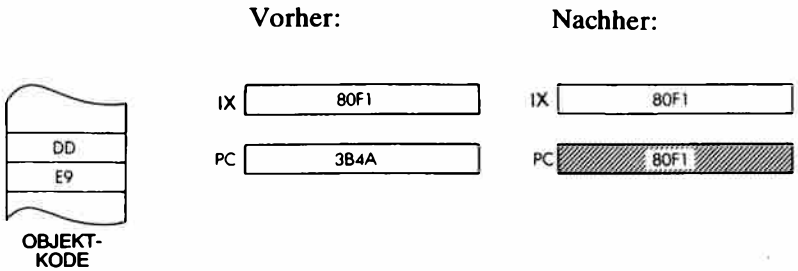
Befehlsablauf: 2 M Zyklen; 8 T Zustände; 4 µsek @2 MHz

Adressierungsart: Implizit.

Flags:

S	Z		H		P/V	N	C	(kein Einfluß)

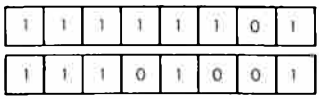
Beispiel: JP (IX)



JP (IY) Sprung zur Adresse (IY)

Funktion: PC ← IY

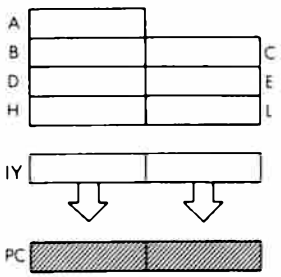
Format:



Byte 1: FD
Byte 2: E9

Beschreibung: Der Inhalt des Registers IY wird in den Befehlszähler geladen. Der nächste Befehl wird von dieser Adresse geholt.

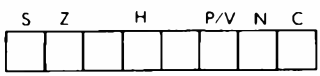
Datenfluß:



Befehlsablauf: 2 M Zyklen; 8 T Zustände; 4 µsek @2 MHz

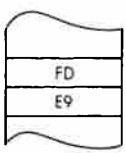
Adressierungsart: Implizit.

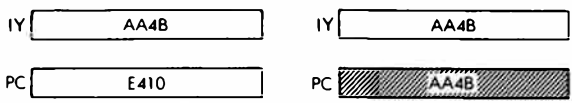
Flags:



(kein Einfluß)

Beispiel: JP (IY)

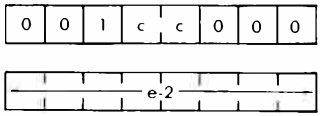




JR cc, e Relativer bedingter Sprung um e.

Funktion: Wenn cc erfüllt, dann PC←PC + e

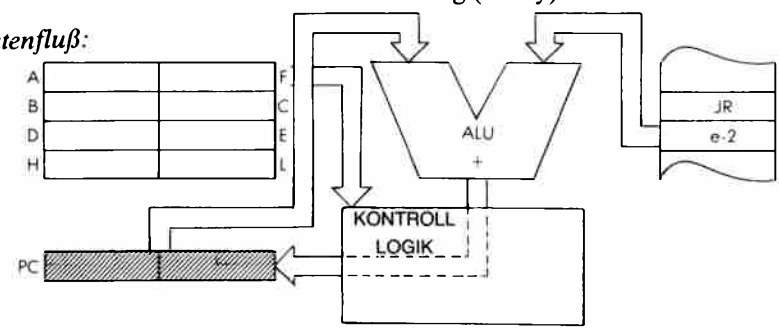
Format:



Beschreibung: Ist die angegebene Bedingung erfüllt, dann wird der Offset in Zweierkomplement-Arithmetik zum Befehlszähler addiert, so daß Sprünge vorwärts und rückwärts möglich sind. Der Offset wird addiert zu PC + 2 (nach dem Sprung). Deshalb ist der effektive Offset −126 bis 129 Byte. Der Assembler subtrahiert automatisch 2 vom Quelloffset, um den Hexcode zu erzeugen. Ist die Bedingung nicht erfüllt, dann wird der Offset nicht beachtet, und die Befehlsausführung wird in der normalen Reihenfolge fortgeführt.
cc kann sein:

NZ – 00	nicht Null
.Z – 01	Null
NC – 10	kein Übertrag (Carry)
C – 11	Übertrag (Carry)

Datenfluß:



Befehlsablauf:

	M Zyklen	*T Zustände*	*µsek @ 2 MHz:*
Bedingung erfüllt:	3	12	6
Bedingung nicht erfüllt:	2	7	3.5

Adressierungsart: Unmittelbar.

Byte Kode:

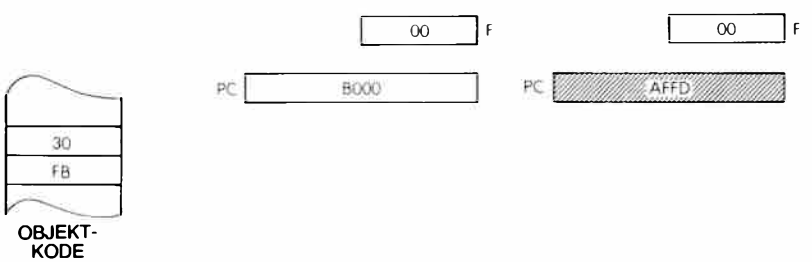
cc:	NZ	Z	NC	C
	20	28	30	38

Flags:

S	Z		H		P/V	N	C

(kein Einfluß)

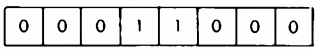
Beispiel: JR NC, $ – 3 $ = current PC



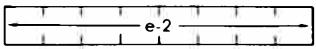
JR e Relativer Sprung um e.

Funktion: PC ← PC + e

Format:



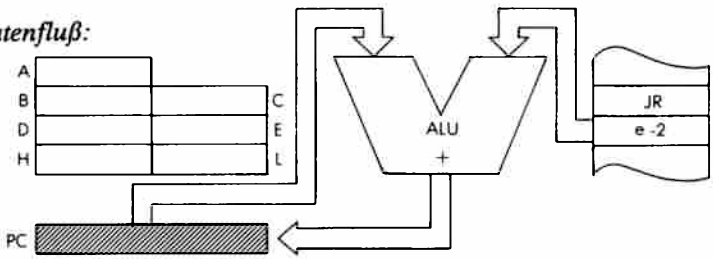
Byte 1: 18



Byte 2: Offset

Beschreibung: Der gegebene Offset wird in Zweierkomplement-Arithmetik zum Befehlszähler addiert, so daß Sprünge vorwärts und rückwärts möglich sind. Der Offset wird addiert zu PC + 2 (nach dem Sprung). Deshalb ist der effektive Offset −126 bis 129 Byte. Der Assembler subtrahiert automatisch 2 vom Quelloffset, um den Hexcode zu erzeugen.

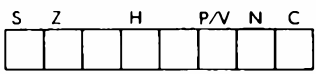
Datenfluß:



Befehlsablauf: 3 M Zyklen; 12 T Zustände; 6 μsek @ 2 MHz

Adressierungsart: Unmittelbar.

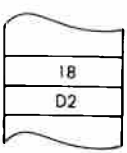
Flags:

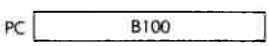


(kein Einfluß)

Beispiel: JR D4

Vorher: Nachher:

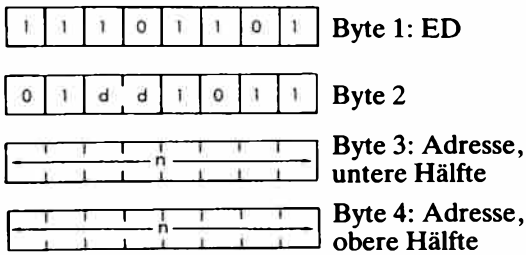




LD dd, (nn) Lade das Registerpaar dd aus der durch nn adressierten Speicherzelle.

Funktion: $dd_{unten} \leftarrow (nn); dd_{oben} \leftarrow (nn + 1)$

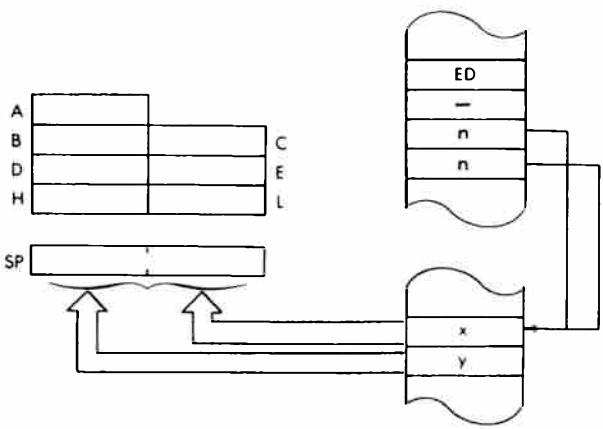
Format:



Beschreibung: Der Inhalt der Speicherzelle, die durch die Speicherinhalte unmittelbar hinter dem Opcode adressiert wird, wird in die untere Hälfte des angegebenen Registerpaars geladen. Der Inhalt der Speicherzelle dahinter kommt in die obere Hälfte. Das untere Byte der Adresse nn folgt unmittelbar auf den Opcode.
dd kann sein:

BC – 00	HL – 10
DE – 01	SP – 11

Datenfluß:



Befehlsablauf: 6 M Zyklen; 20 T Zustände; 10 μsek @ 2 MHz

Adressierungsart: Direkt.

Byte Kode:

dd:	BC	DE	HL	SP
	4B	5B	6B	7B

Flags:

S	Z		H		P/V	N	C

(kein Einfluß)

Beispiel: LD DE, (5021)

Vorher:

D	DBE2	E

Nachher:

D	30F4	E

OBJEKT-KODE

ED
5B
21
50

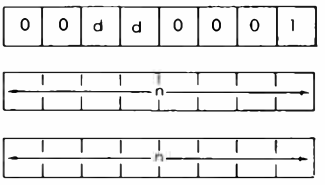
5021	F4
5022	30

5021	F4
5022	30

LD dd, nn Lade das Registerpaar dd mit den unmittelbaren Daten nn.

Funktion: dd ← nn

Format:



Byte 1

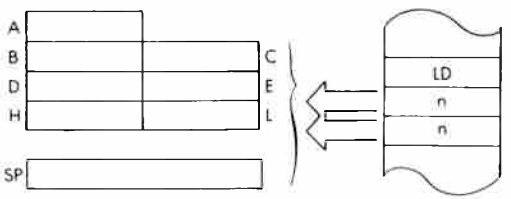
Byte 2: unmittelbare Daten, untere Hälfte

Byte 3: unmittelbare Daten, obere Hälfte

Beschreibung: Der Inhalt der beiden Speicherzellen, die unmittelbar hinter dem Opcode stehen, wird in das angegebene Registerpaar geladen. Die untere Hälfte der Daten folgt unmittelbar auf den Opcode.
dd kann sein:

BC – 00 HL – 10
DE – 01 SP – 11

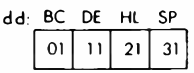
Datenfluß:



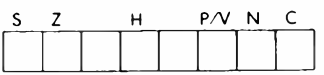
Befehlsablauf: 3 M Zyklen; 10 T Zustände; 5 µsek @ 2 MHz

Adressierungsart: Unmittelbar.

Byte Kode:

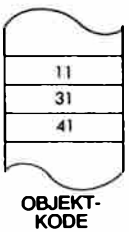


Flags:

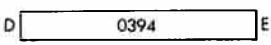


(kein Einfluß)

Beispiel: LD DE, 4131



Vorher:

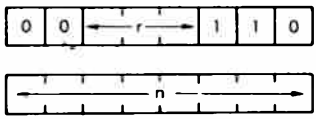


Nachher:

LD r, n Lade das Register r mit den unmittelbaren Daten n.

Funktion: r ← n

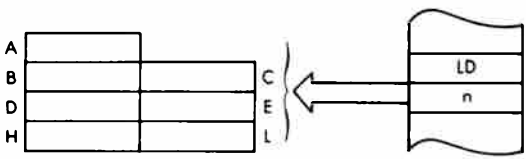
Format:



Beschreibung: Der Inhalt der Speicherzelle, die unmittelbar auf den Opcode folgt, wird in das angegebene Register r geladen.
r kann sein:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

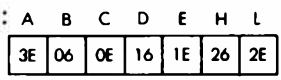
Datenfluß:



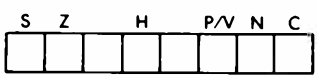
Befehlsablauf: 2 M Zyklen; 7 T Zustände; 3,5 µsek @ 2 MHz

Adressierungsart: Unmittelbar.

Byte Kode:

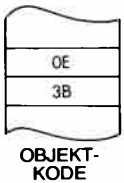


Flags:



(kein Einfluß)

Beispiel: LD C, 3B



Vorher: C 01

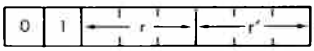
Nachher: C 3B

LD r, r'

Lade das Register r aus dem Register r'.

Funktion: r ← r'

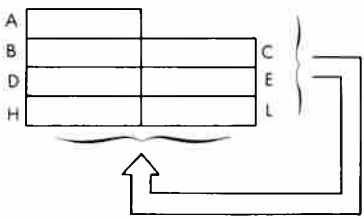
Format:



Beschreibung: Der Inhalt des Quellregisters wird in das angegebene Zielregister geladen.
r und r' können sein:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Datenfluß:



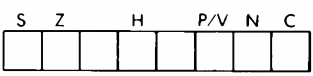
Befehlsablauf: 1 M Zyklus; 4 T Zustände; 2 μsek @ 2 MHz

Adressierungsart: Implizit.

Byte Kode:

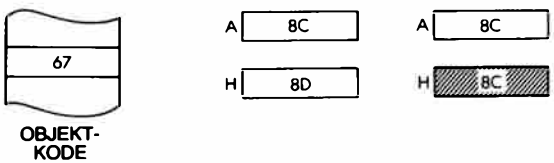
(Ziel) \ (Quelle)	A	B	C	D	E	H	L
A	7F	78	79	7A	7B	7C	7D
B	47	40	41	42	43	44	45
C	4F	48	49	4A	4B	4C	4D
D	57	50	51	52	53	54	55
E	5F	58	59	5A	5B	5C	5D
H	67	60	61	62	63	64	65
L	6F	68	69	6A	6B	6C	6D

Flags:



(kein Einfluß)

Beispiel: LD H, A



LD (BC), A Lade die indirekt adressierte Speicherzelle (BC) aus dem Akkumulator.

Funktion: (BC) ← A

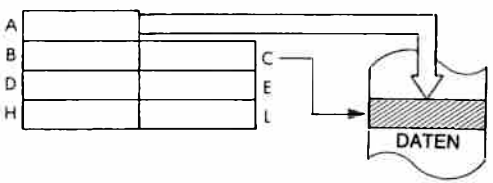
Format:

0	0	0	0	0	0	1	0

02

Beschreibung: Der Inhalt des Akkumulators wird in die durch den Inhalt von BC adressierte Speicherzelle geladen.

Datenfluß:



Befehlsablauf: 2 M Zyklen; 7 T Zustände; 3,5 µsek @ 2 MHz

Adressierungsart: Indirekt.

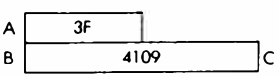
Flags:

S	Z	H	P/V	N	C

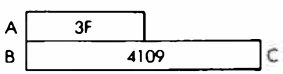
(kein Einfluß)

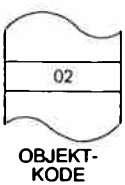
Beispiel: LD (BC), A

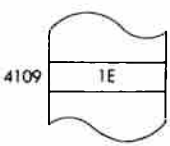
Vorher:

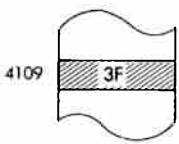


Nachher:









LD (DE), A

Lade die indirekt adressierte Speicherzelle (DE) aus dem Akkumulator.

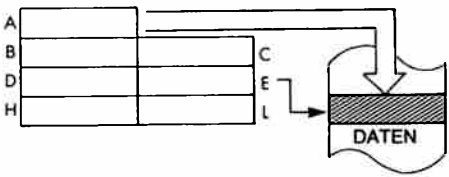
Funktion: (DE) ← A

Format:

0	0	0	1	0	0	1	0	12

Beschreibung: Der Inhalt des Akkumulators wird in die durch den Inhalt von DE adressierte Speicherzelle geladen.

Datenfluß:



Befehlsablauf: 2 M Zyklen; 7 T Zustände; 3,5 µsek @ 2 MHz

Adressierungsart: Indirekt.

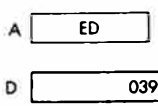
Flags:

S	Z		H		P/V	N	C

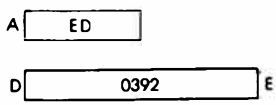
(kein Einfluß)

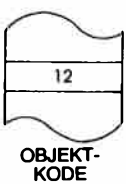
Beispiel: LD (DE), A

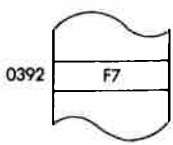
Vorher:

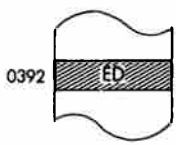


Nachher:





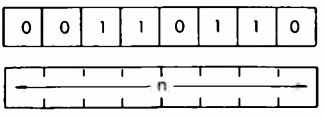




LD (HL), n Lade die unmittelbaren Daten n in die indirekt adressierte Speicherzelle (HL)

Funktion: (HL) ← n

Format:

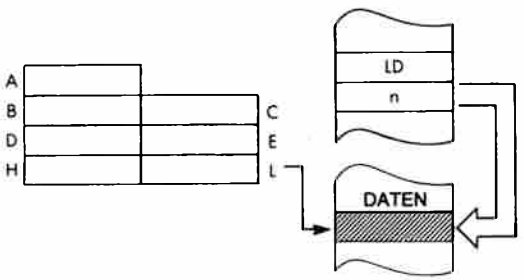


Byte 1: 36

Byte 2: unmittelbare Daten

Beschreibung: Der Inhalt der Speicherzelle, die unmittelbar auf den Opcode folgt, wird in die durch HL adressierte Speicherzelle geladen.

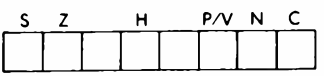
Datenfluß:



Befehlsablauf: 3 M Zyklen; 10 T Zustände; 5 µsek @ 2 MHz

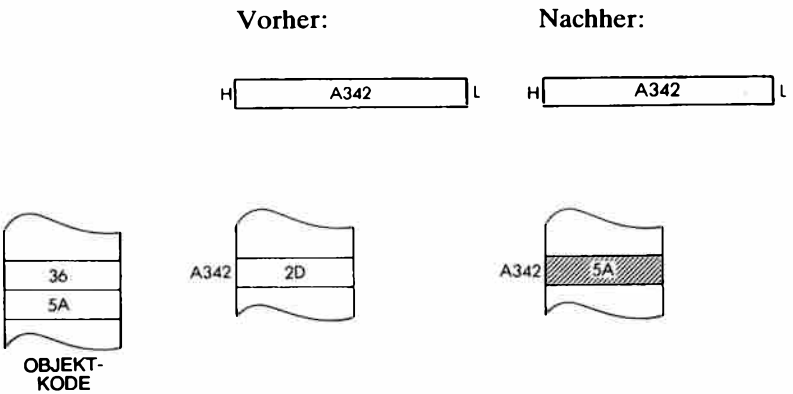
Adressierungsart: Unmittelbar/indirekt.

Flags:



(kein Einfluß)

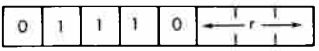
Beispiel: LD (HL), 5A



LD (HL), r Lade die indirekt adressierte Speicherzelle (HL) aus dem Register r.

Funktion: (HL) ← r

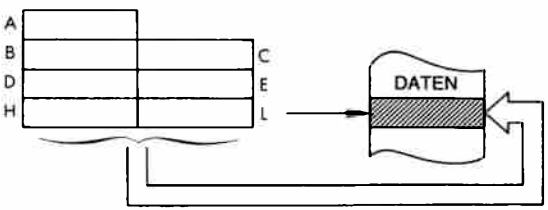
Format:



Beschreibung: Der Inhalt des angegebenen Registers wird in die durch den Inhalt von HL adressierte Speicherzelle geladen.
r kann sein:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Datenfluß:



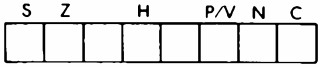
Befehlsablauf: 2 M Zyklen; 7 T Zustände; 3,5 µsek @ 2 MHz

Adressierungsart: Indirekt.

Byte Kode:

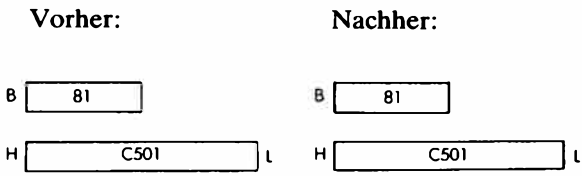
r:	A	B	C	D	E	H	L
	77	70	71	72	73	74	75

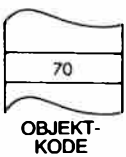
Flags:

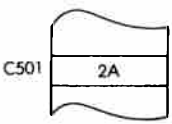


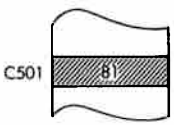
(kein Einfluß)

Beispiel: LD (HL), B





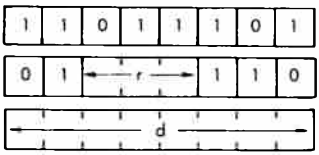




LD r, (IX + d) Lade das Register r indirekt aus der indiziert adressierten Speicherzelle (IX + d).

Funktion: r ← (IX + d)

Format:

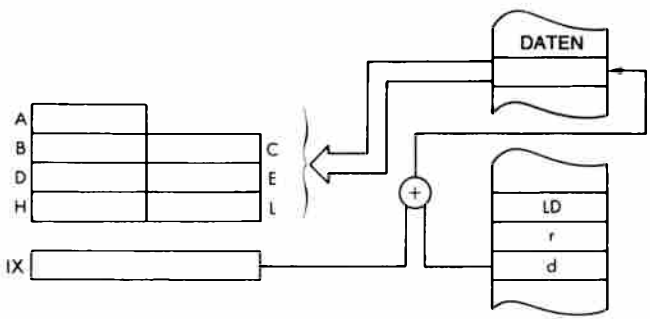


Byte 1: DD
Byte 2
Byte 3: Offset

Beschreibung: Der Inhalt der Speicherzelle, die durch das Register IX plus dem gegebenen Offset adressiert wird, wird ins Register r geladen.
r kann sein:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Datenfluß:



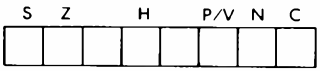
Befehlsablauf: 5 M Zyklen; 19 T Zustände; 9,5 μsek @ 2 MHz

Adressierungsart: Indiziert.

Byte Kode:

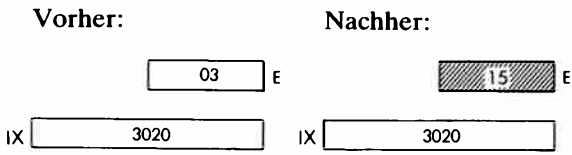
r:	A	B	C	D	E	H	L	
DD-	7E	46	4E	56	5E	66	6E	-d

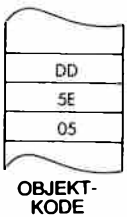
Flags:

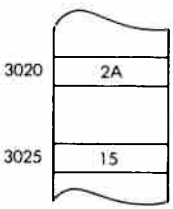


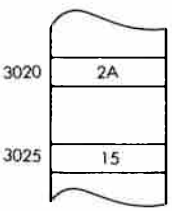
(kein Einfluß)

Beispiel: LD E, (IX + 5)





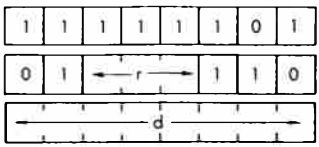




LD r, (IY + d) Lade das Register r indirekt aus der indiziert adressierten Speicherzelle (IY + d).

Funktion: r ← (IY + d)

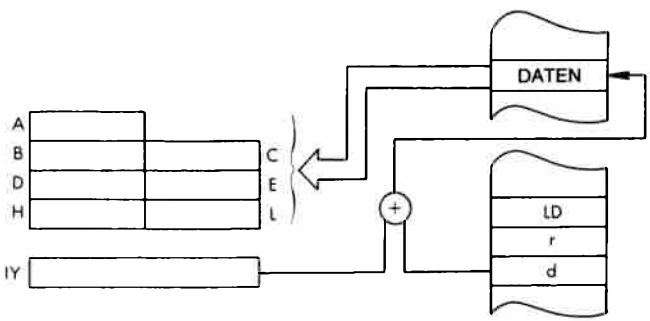
Format:



Beschreibung: Der Inhalt der Speicherzelle, die durch das Register IY plus dem gegebenen Offset adressiert wird, wird ins Register r geladen.
r kann sein:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

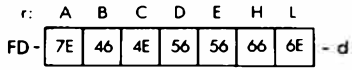
Datenfluß:



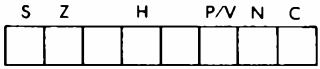
Befehlsablauf: 5 M Zyklen; 19 T Zustände; 9,5 µsek @ 2 MHz

Adressierungsart: Indiziert.

Byte Kode:



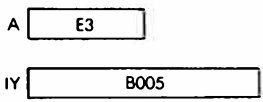
Flags:



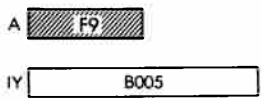
(kein Einfluß)

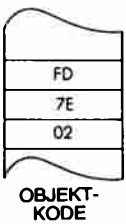
Beispiel: LD A, (IY + 2)

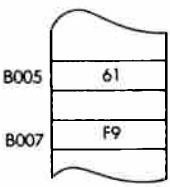
Vorher:

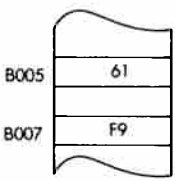


Nachher:





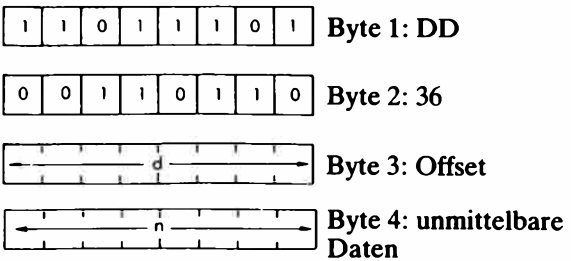




LD (IX + d), n Lade die indiziert adressierte Speicherzelle (IX + d) mit den unmittelbaren Daten n.

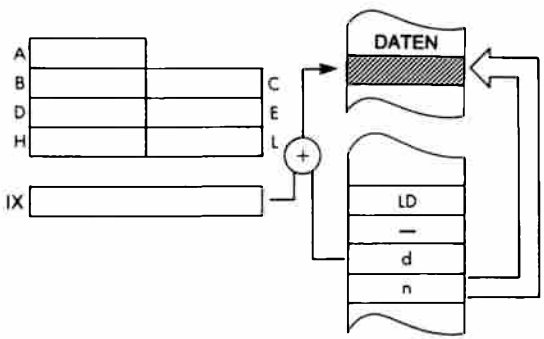
Funktion: (IX + d) ← n

Format:



Beschreibung: Der Inhalt der Speicherzelle, die unmittelbar auf den Opcode folgt, wird in die Speicherzelle geladen, die durch das Register IX plus dem gegebenen Offset adressiert wird.

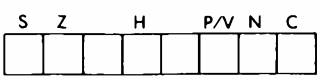
Datenfluß:



Befehlsablauf: 5 M Zyklen; 19 T Zustände; 9,5 µsek @ 2 MHz

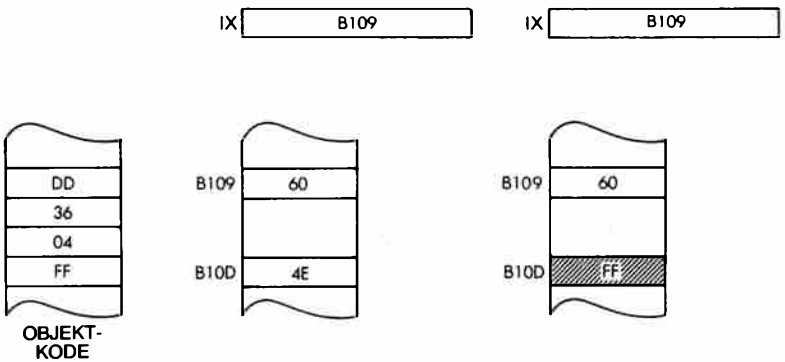
Adressierungsart: Indiziert/unmittelbar.

Flags:



(kein Einfluß)

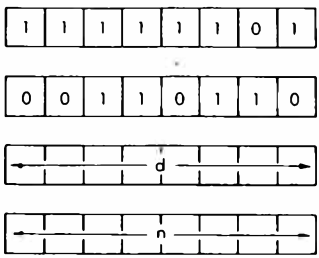
Beispiel: LD (IX + 4), FF



LD (IY + d), n Lade die indiziert adressierte Speicherzelle (IY + d) mit den unmittelbaren Daten n.

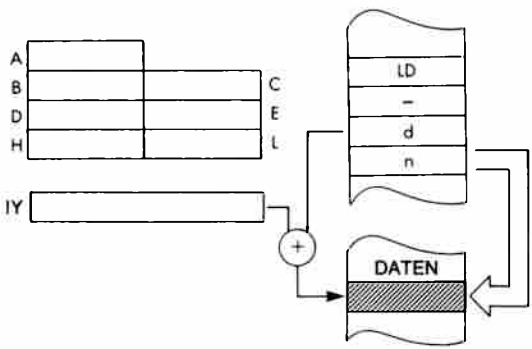
Funktion: (IY + d) ← n

Format:



Beschreibung: Der Inhalt der Speicherzelle, die unmittelbar auf den Opcode folgt, wird in die Speicherzelle geladen, die durch das Register IY plus dem gegebenen Offset adressiert wird.

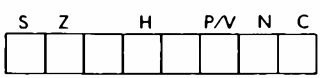
Datenfluß:



Befehlsablauf: 5 M Zyklen; 19 T Zustände; 9,5 µsek @ 2 MHz

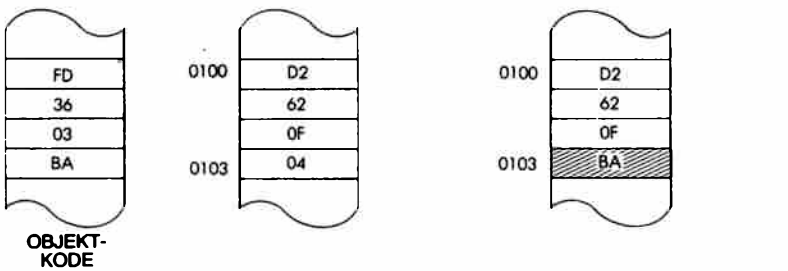
Adressierungsart: Indiziert/unmittelbar.

Flags:



(kein Einfluß)

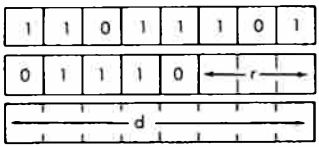
Beispiel: LD (IY + 3), BA



LD (IX + d),r Lade die indiziert adressierte Speicherzelle (IX + d) aus dem Register r.

Funktion: (IX + d) ← r

Format:

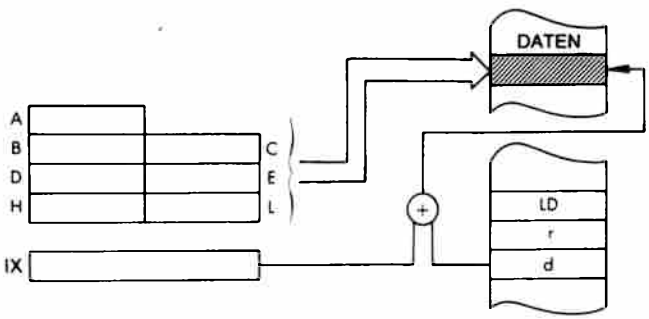


Byte 1: DD
Byte 2
Byte 3: Offset

Beschreibung: Der Inhalt des angegebenen Registers wird in die Speicherzelle geladen, die durch das Register IX plus dem gegebenen Offset adressiert wird.
r kann sein:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

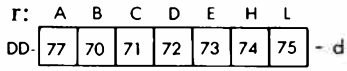
Datenfluß:



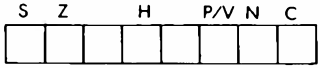
Befehlsablauf: 5 M Zyklen; 19 T Zustände; 9,5 µsek @ 2 MHz

Adressierungsart: Indiziert.

Byte Kode:



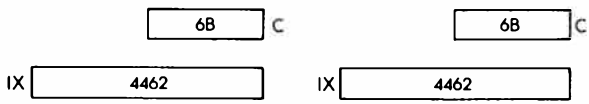
Flags:

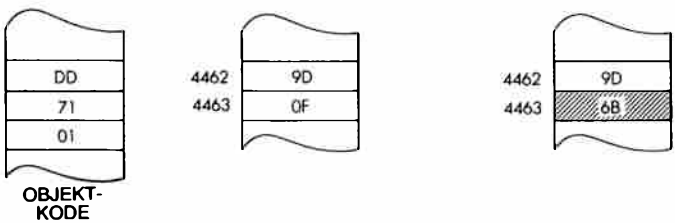


(kein Einfluß)

Beispiel: LD (IX + 1), C

Vorher: Nachher:

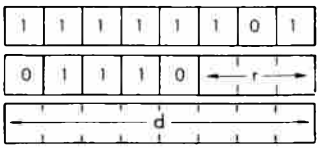




LD (IY + d), r Lade die indiziert adressierte Speicherzelle (IY + d) aus dem Register r.

Funktion: (IY + d) ← r

Format:



Byte 1: FD

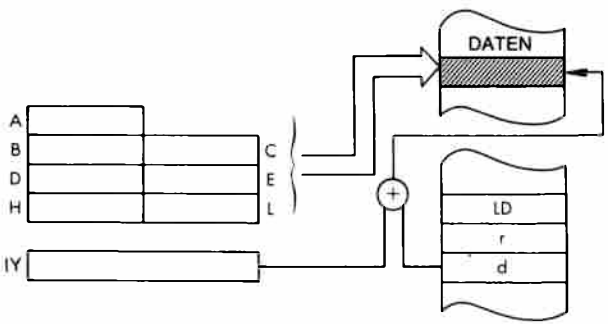
Byte 2

Byte 3: Offset

Beschreibung: Der Inhalt des angegebenen Registers wird in die Speicherzelle geladen, die durch das Register IY plus dem gegebenen Offset adressiert wird.
r kann sein:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Datenfluß:



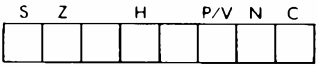
Befehlsablauf: 5 M Zyklen; 19 T Zustände; 9,5 µsek @ 2 MHz

Adressierungsart: Indiziert.

Byte Kode:

r:	A	B	C	D	E	H	L	
FD-	77	70	71	72	73	74	75	-d

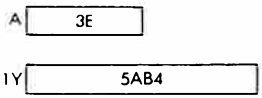
Flags:



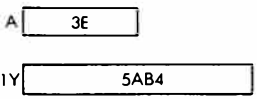
(kein Einfluß)

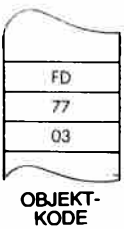
Beispiel: LD (IY + 3), A

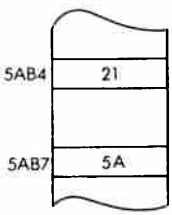
Vorher:

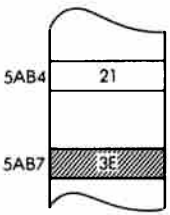


Nachher:





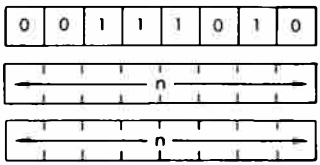




LD A, (nn) Lade den Akkumulator aus der Speicherstelle (nn).

Funktion: A ← (nn)

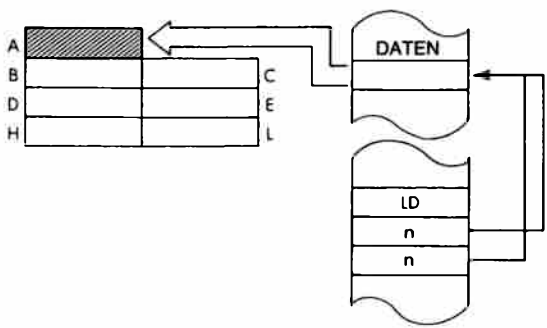
Format:



Byte 1: 3A
Byte 2: Adresse, untere Hälfte
Byte 3: Adresse, obere Hälfte

Beschreibung: Der Inhalt der Speicherzelle, die durch den Inhalt der beiden Bytes nach dem Opcode adressiert wird, wird in den Akkumulator geladen. Das untere Byte der Adresse folgt unmittelbar auf den Opcode.

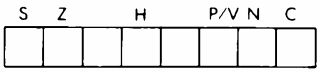
Datenfluß:



Befehlsablauf: 4 M Zyklen; 13 T Zustände; 6,5 µsek @ 2 MHz

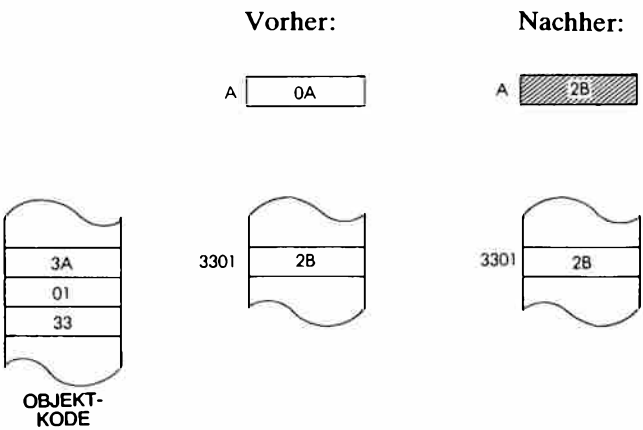
Adressierungsart: Direkt.

Flags:



(kein Einfluß)

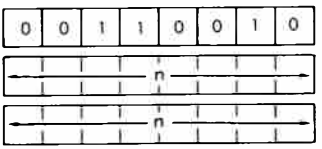
Beispiel: LD A, (3301)



LD (nn), A Lade die direkt adressierte Speicherstelle (nn) aus dem Akkumulator.

Funktion: (nn) ← A

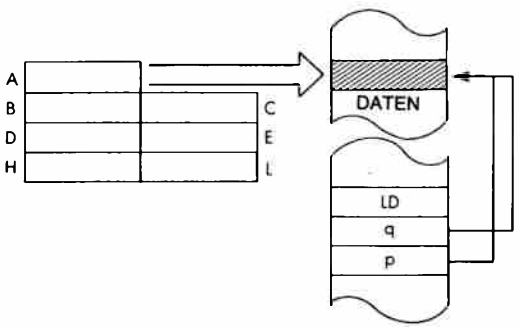
Format:



Byte 1: 32
Byte 2: Adresse, untere Hälfte
Byte 3: Adresse, obere Hälfte

Beschreibung: Der Inhalt der Speicherzelle, die durch den Inhalt der beiden Bytes nach dem Opcode adressiert wird, wird aus dem Akkumulator geladen. Das untere Byte der Adresse folgt unmittelbar auf den Opcode.

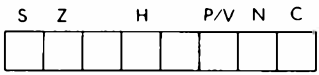
Datenfluß:



Befehlsablauf: 4 M Zyklen; 13 T Zustände; 6,5 μsek @ 2 MHz

Adressierungsart: Direkt.

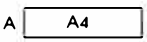
Flags:



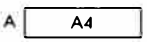
(kein Einfluß)

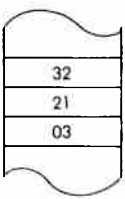
Beispiel: LD (0321), A

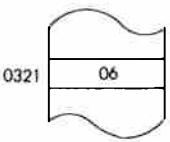
Vorher:

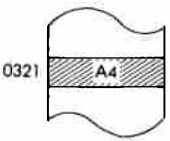


Nachher:





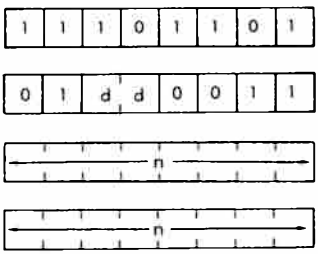




LD (nn), dd Lade die durch nn adressierten Speicherstellen aus dem Registerpaar dd.

Funktion: $(nn) \leftarrow dd_{unten}; (nn + 1) \leftarrow dd_{oben}$

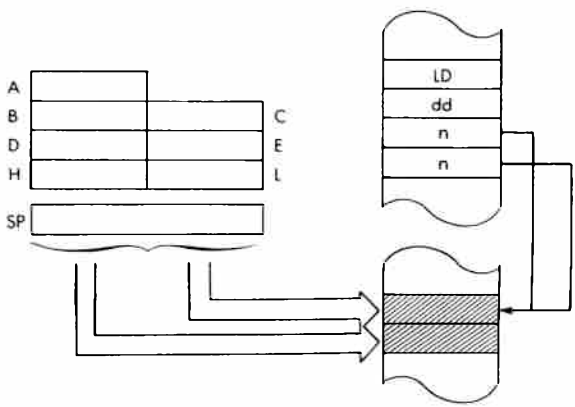
Format:



Beschreibung: Die untere Hälfte des angegebenen Registerpaares wird in die Speicherzelle geladen, die durch die Speicherstellen adressiert ist, die unmittelbar auf den Opcode folgen. Die obere Hälfte wird aus der Speicherzelle dahinter geladen. Die untere Hälfte der Adresse nn erscheint unmittelbar hinter dem Opcode.
dd kann sein:

BC – 00	HL – 10
DE – 01	SP – 11

Datenfluß:



Befehlsablauf: 6 M Zyklen; 20 T Zustände; 10 μsek @ 2 MHz

Adressierungsart: Direkt.

Byte Kode:

dd:	BC	DE	HL	SP
ED-	43	53	63	73

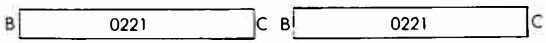
Flags:

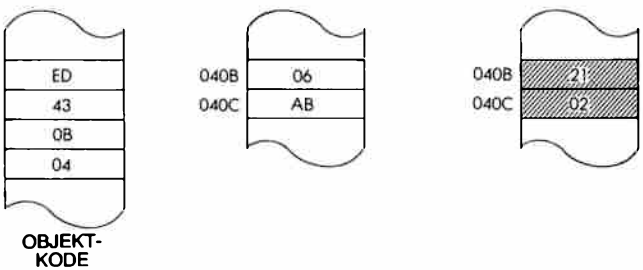
S	Z		H		P/V	N	C

(kein Einfluß)

Beispiel: LD (040B), BC

Vorher: Nachher:

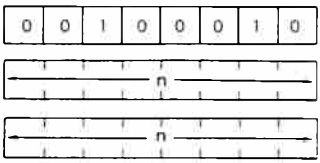




LD (nn), HL Lade die durch nn adressierten Speicherstellen aus dem Register HL.

Funktion: (nn) ← L; (nn + 1) ← H

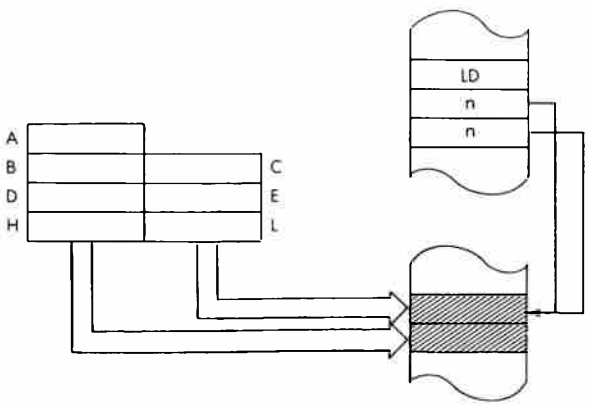
Format:



Byte 1: 22
Byte 2: Adresse, untere Hälfte
Byte 3: Adresse, obere Hälfte

Beschreibung: Der Inhalt des Registers L wird in die Speicherzelle geladen, die durch die Speicherstellen adressiert ist, die unmittelbar auf den Opcode folgen. Das Register H wird aus der Speicherzelle dahinter geladen. Die untere Hälfte der Adresse nn erscheint unmittelbar hinter dem Opcode.

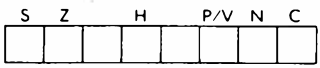
Datenfluß:



Befehlsablauf: 5 M Zyklen; 16 T Zustände; 8 μsek @ 2 MHz

Adressierungsart: Direkt.

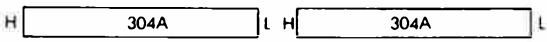
Flags:

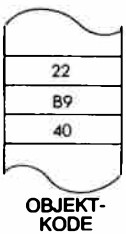


(kein Einfluß)

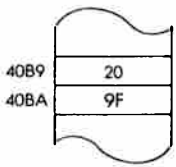
Beispiel: LD (40B9), HL

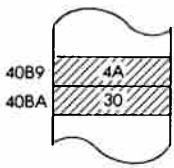
Vorher: **Nachher:**





OBJEKT-KODE

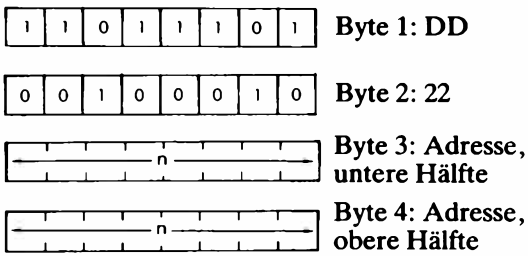




LD (nn), IX Lade die durch nn adressierten Speicherstellen aus dem Register IX.

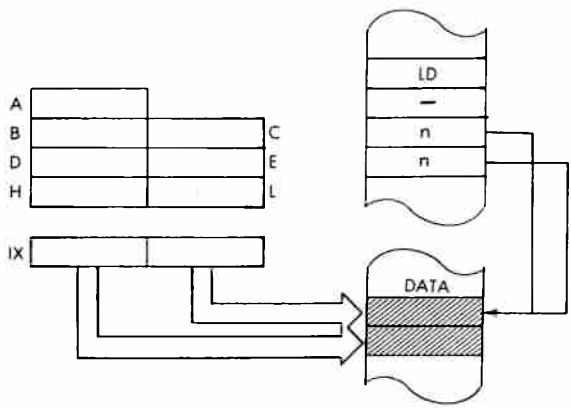
Funktion: $(nn) \leftarrow IX_{unten}; (nn + 1) \leftarrow IX_{oben}$

Format:



Beschreibung: Die untere Hälfte des Registers IX wird in die Speicherzelle geladen, die durch die Speicherstellen adressiert ist, die unmittelbar auf den Opcode folgen. Die obere Hälfte wird aus der Speicherzelle dahinter geladen. Die untere Hälfte der Adresse nn erscheint unmittelbar hinter dem Opcode.

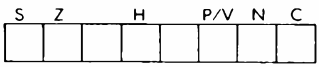
Datenfluß:



Befehlsablauf: 6 M Zyklen; 20 T Zustände; 10 μsek @ 2 MHz

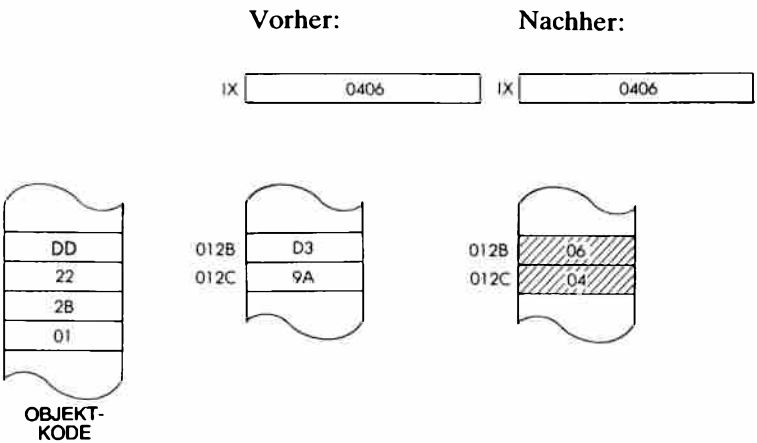
Adressierungsart: Direkt.

Flags:



(kein Einfluß)

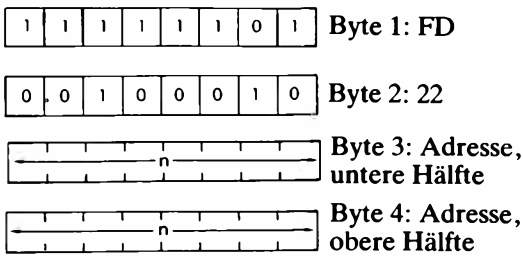
Beispiel: LD (012B), IX



LD (nn), IY Lade die durch nn adressierten Speicherstellen aus dem Register IY.

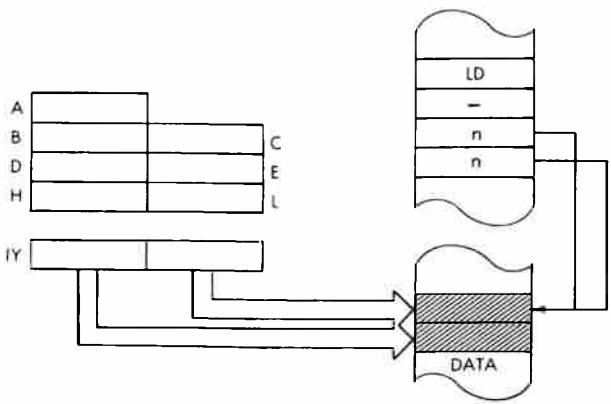
Funktion: $(nn) \leftarrow IY_{unten}; (nn + 1) \leftarrow IY_{oben}$

Format:



Beschreibung: Die untere Hälfte des Registers IY wird in die Speicherzelle geladen, die durch die Speicherstellen adressiert ist, die unmittelbar auf den Opcode folgen. Die obere Hälfte wird aus der Speicherzelle dahinter geladen. Die untere Hälfte der Adresse nn erscheint unmittelbar hinter dem Opcode.

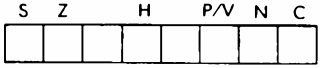
Datenfluß:



Befehlsablauf: 6 M Zyklen; 20 T Zustände; 10 μsek @ 2 MHz

Adressierungsart: Direkt.

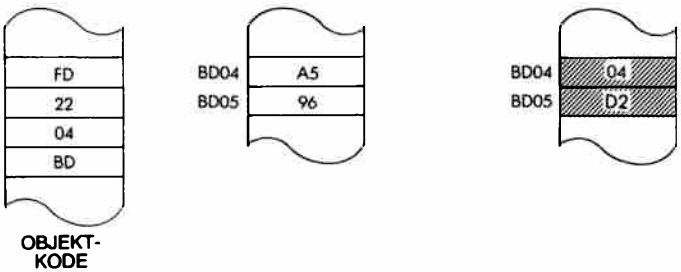
Flags:



(kein Einfluß)

Beispiel: **LD (BD04), IY**

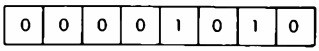
Vorher: **Nachher:**



LD A, (BC) Lade den Akkumulator aus der durch das Registerpaar BC indirekt adressierten Speicherstelle.

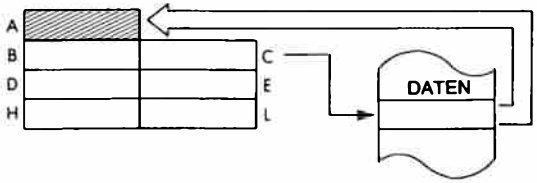
Funktion: A ← (BC)

Format:



Beschreibung: Der Inhalt der Speicherzelle, die durch den Inhalt des Registerpaares BC adressiert wird, wird in den Akkumulator geladen.

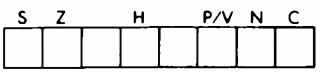
Datenfluß:



Befehlsablauf: 2 M Zyklen; 7 T Zustände; 3,5 µsek @ 2 MHz

Adressierungsart: Indirekt.

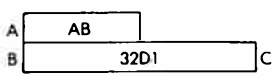
Flags:

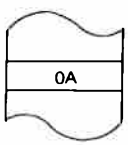


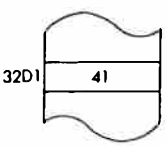
(kein Einfluß)

Beispiel: LD A, (BC)

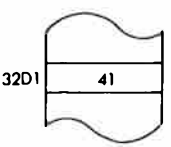
Vorher:







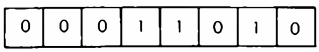
Nachher:



LD A, (DE) Lade den Akkumulator aus der durch das Registerpaar DE indirekt adressierten Speicherstelle.

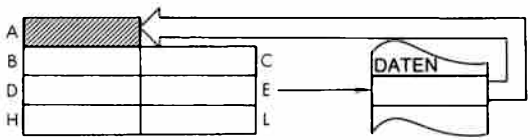
Funktion: A ← (DE)

Format:



Beschreibung: Der Inhalt der Speicherzelle, die durch den Inhalt des Registerpaares DE adressiert wird, wird in den Akkumulator geladen.

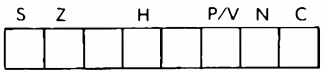
Datenfluß:



Befehlsablauf: 2 M Zyklen; 7 T Zustände; 3,5 µsek @ 2 MHz

Adressierungsart: Indirekt.

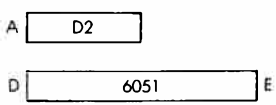
Flags:



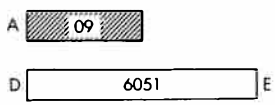
(kein Einfluß)

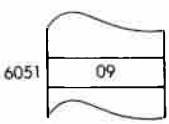
Beispiel: LD A, (DE)

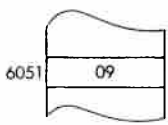
Vorher:



Nachher:





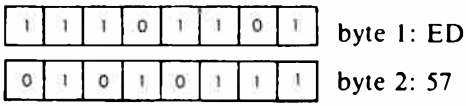


LD A, I

Lade den Akkumulator aus dem Interruptvektor-Register I.

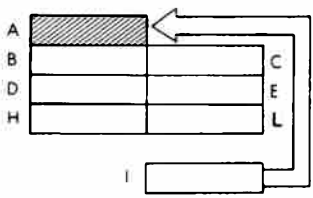
Funktion: A ← I

Format:



Beschreibung: Der Inhalt des Interruptvektor-Registers I wird in den Akkumulator geladen.

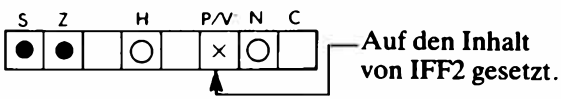
Datenfluß:



Befehlsablauf: 2 M Zyklen; 9 T Zustände; 4,5 µsek @ 2 MHz

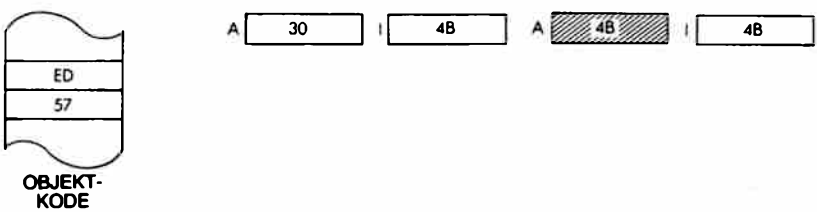
Adressierungsart: Implizit.

Flags:



Beispiel: LD A, I

Vorher: Nachher:



LD I, A Lade das Interrupt-Register I aus dem Akkumulator

Funktion: I ← A

Format:

1	1	1	0	1	1	0	1

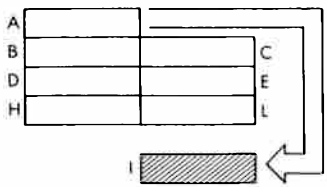
Byte 1: ED

0	1	0	0	0	1	1	1

Byte 2: 47

Beschreibung: Der Inhalt des Akkumulators wird in das Interruptvektor-Register I geladen.

Datenfluß:



Befehlsablauf: 2 M Zyklen; 9 T Zustände; 4,5 µsek @ 2 MHz

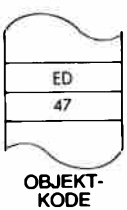
Adressierungsart: Implizit.

Flags:

S	Z		H		P/V	N	C

(kein Einfluß)

Beispiel: LD I, A



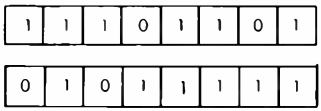
Vorher: A 06 I D2

Nachher: A 06 I 06

LD A, R Lade den Akkumulator aus dem Memory-Refresh-Register R.

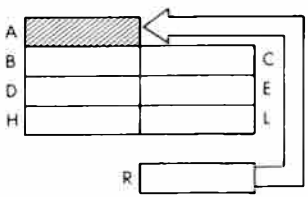
Funktion: A ← R

Format:



Beschreibung: Der Inhalt des Memory-Refresh-Registers wird in den Akkumulator geladen.

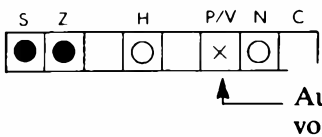
Datenfluß:



Befehlsablauf: 2 M Zyklen; 9 T Zustände; 4,5 μsek @ 2 MHz

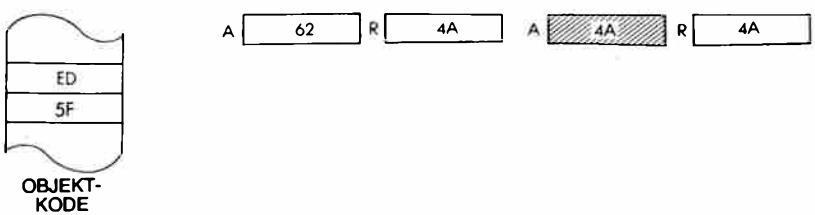
Adressierungsart: Implizit.

Flags:



Beispiel: LD A, R

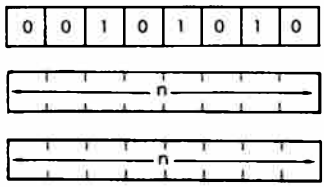
Vorher: Nachher:



LD HL, (nn) Lade das Register HL aus der Speicherzelle nn.

Funktion: L ← (nn); H ← (nn + 1)

Format:



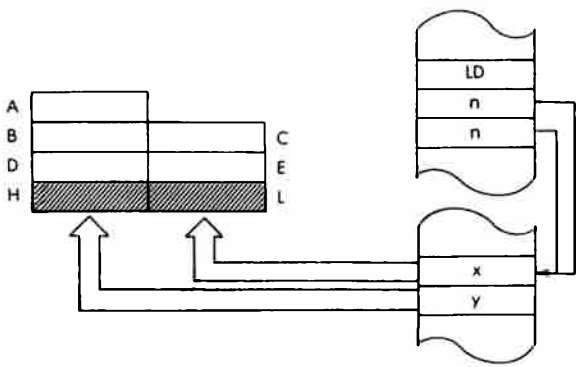
Byte 1: 2A

Byte 2: Adresse, untere Hälfte

Byte 3: Adresse, obere Hälfte

Beschreibung: Der Inhalt der Speicherzelle, die durch die Speicherstellen adressiert ist, die unmittelbar auf den Opcode folgen, wird ins Register L geladen. Das Register H wird aus der Speicherzelle dahinter geladen. Die untere Hälfte der Adresse nn erscheint unmittelbar hinter dem Opcode.

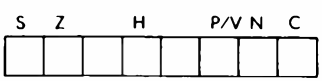
Datenfluß:



Befehlsablauf: 5 M Zyklen; 16 T Zustände; 8 µsek @ 2 MHz

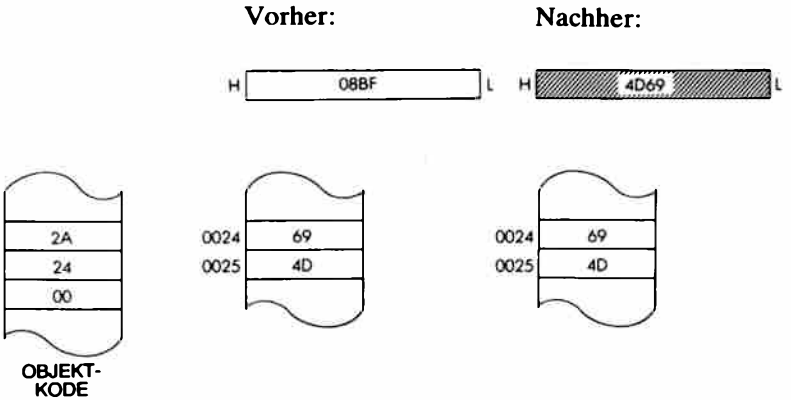
Adressierungsart: Direkt.

Flags:



(kein Einfluß)

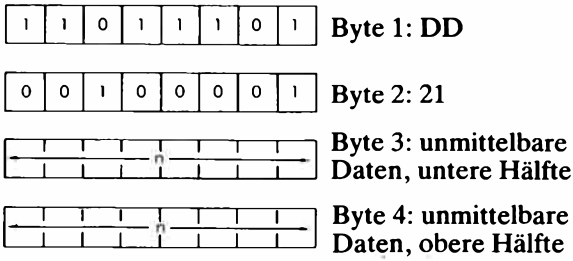
Beispiel: LD HL, (0024)



LD IX, nn Lade das Register IX mit den unmittelbaren Daten nn.

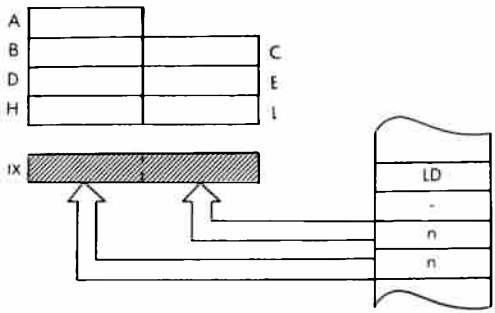
Funktion: IX ← nn

Format:



Beschreibung: Der Inhalt der Speicherzellen, die unmittelbar auf den Opcode folgen, wird in das Register IX geladen. Das untere Byte erscheint unmittelbar hinter dem Opcode.

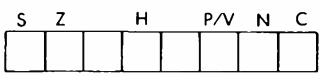
Datenfluß:



Befehlsablauf: 4 M Zyklen; 14 T Zustände; 7 μsek @ 2 MHz

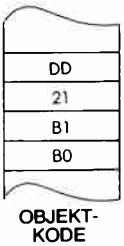
Adressierungsart: Unmittelbar.

Flags:

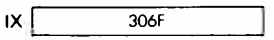


(kein Einfluß)

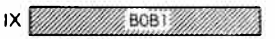
Beispiel: LD IX, B0B 1



Vorher:



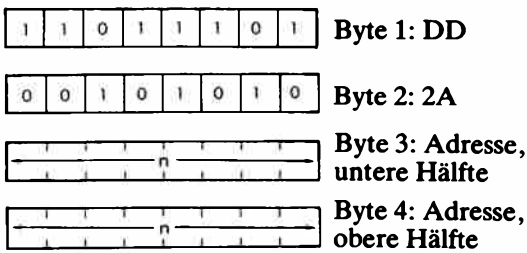
Nachher:



LD IX, (nn) Lade das Register IX aus der Speicherzelle nn.

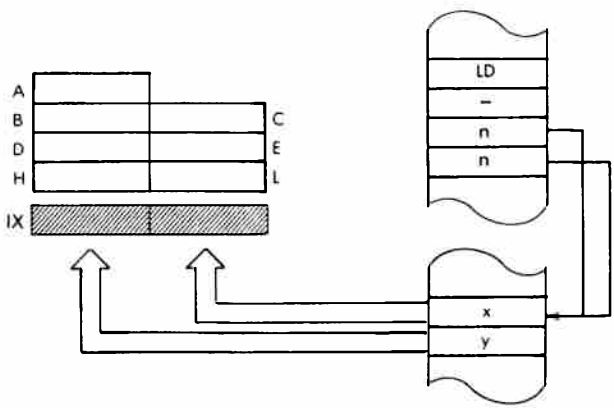
Funktion: $IX_{unten} \leftarrow (nn); IX_{oben} \leftarrow (nn + 1)$

Format:



Beschreibung: Der Inhalt der Speicherzelle, die durch die Speicherstellen adressiert ist, die unmittelbar auf den Opcode folgen, wird in die untere Hälfte des Registers IX geladen. Die obere Hälfte wird aus der Speicherzelle dahinter geladen. Die untere Hälfte der Adresse nn erscheint unmittelbar hinter dem Opcode.

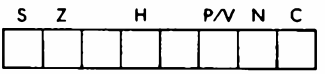
Datenfluß:



Befehlsablauf: 6 M Zyklen; 20 T Zustände; 10 µsek @ 2 MHz

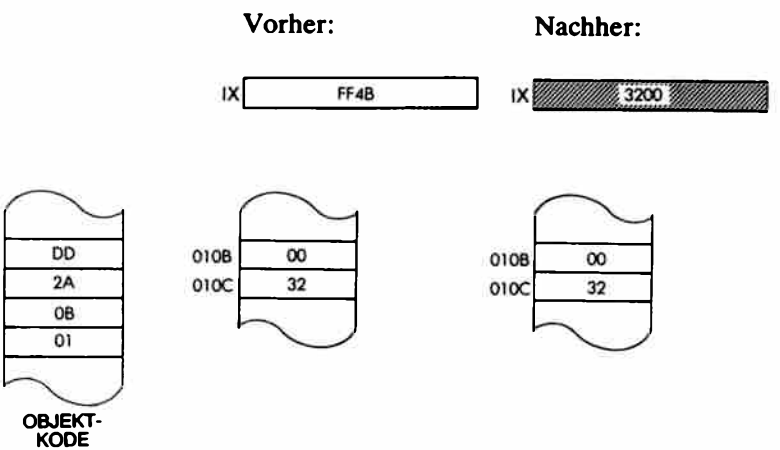
Adressierungsart: Direkt.

Flags:



(kein Einfluß)

Beispiel: LD IX, (010B)

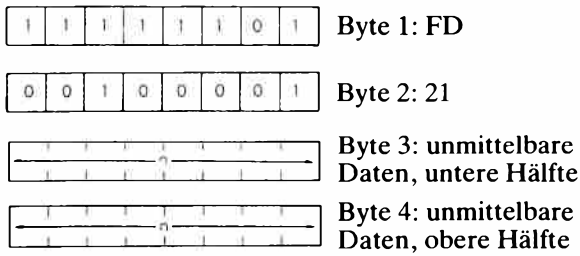


LD IY, nn

Lade das Register IY mit den unmittelbaren Daten nn.

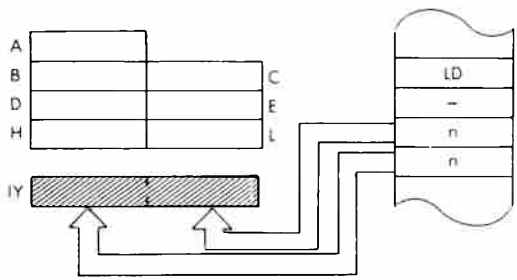
Funktion: IY ← nn

Format:



Beschreibung: Der Inhalt der Speicherzellen, die unmittelbar auf den Opcode folgen, wird in das Register IY geladen. Das untere Byte erscheint unmittelbar hinter dem Opcode.

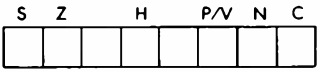
Datenfluß:



Befehlsablauf: 4 M Zyklen; 14 T Zustände; 7 μsek @ 2 MHz

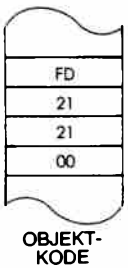
Adressierungsart: Unmittelbar.

Flags:

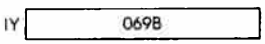


(kein Einfluß)

Beispiel: LD IY, 21



Vorher:



Nachher:

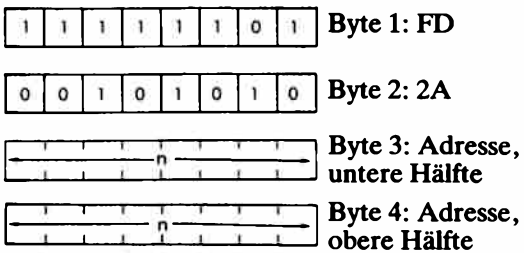
LD IY, (nn)

Lade das Register IY aus der Speicherzelle nn.

Funktion:

$IY_{unten} \leftarrow (nn); IY_{oben} \leftarrow (nn + 1)$

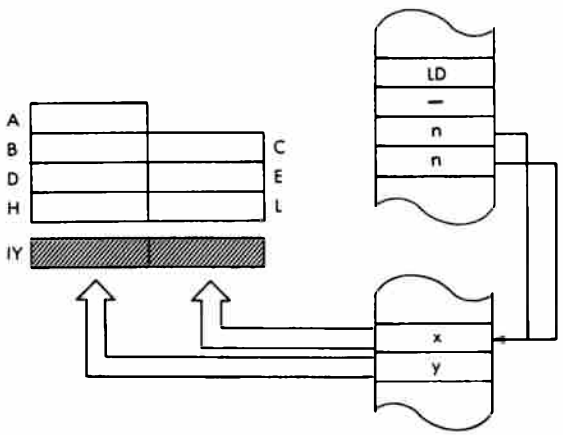
Format:



Beschreibung:

Der Inhalt der Speicherzelle, die durch die Speicherstellen adressiert ist, die unmittelbar auf den Opcode folgen, wird in die untere Hälfte des Registers IY geladen. Die obere Hälfte wird aus der Speicherzelle dahinter geladen. Die untere Hälfte der Adresse nn erscheint unmittelbar hinter dem Opcode.

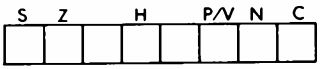
Datenfluß:



Befehlsablauf: 6 M Zyklen; 20 T Zustände; 10 μsek @ 2 MHz

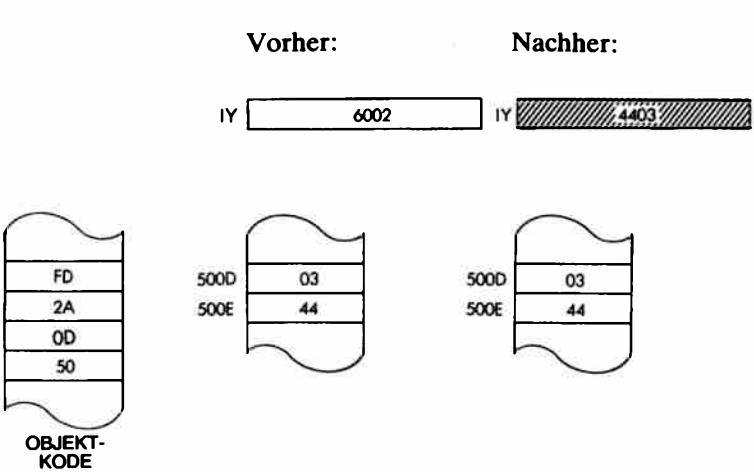
Adressierungsart: Direkt.

Flags:



(kein Einfluß)

Beispiel: LD IY, (500D)



LD R,A

Lade das Memory-Refresh-Register aus dem Akkumulator.

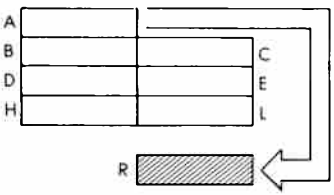
Funktion: R ← A

Format:

1	1	1	0	1	1	0	1	byte 1: ED
0	1	0	0	1	1	1	1	byte 2: 4F

Beschreibung: Der Inhalt des Akkumulators wird in das Memory-Refresh-Register geladen.

Datenfluß:



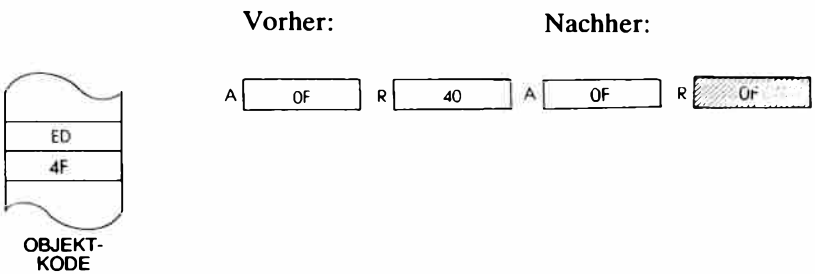
Befehlsablauf: *2 M Zyklen; 9 T Zustände; 4,5 µsek @2 MHz*

Adressierungsart: Implizit.

Flags:

S	Z		H		P/V	N	C	(kein Einfluß)

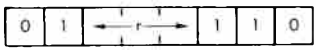
Beispiel: LD R, A



LD r, (HL) Lade das Register r indirekt aus der Speicherstelle (HL).

Funktion: r ← (HL)

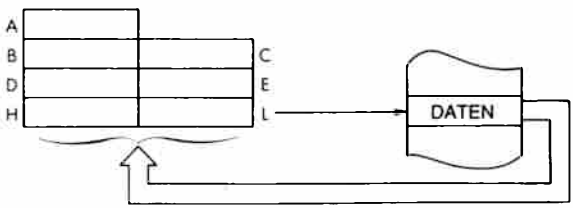
Format:



Beschreibung: Der Inhalt der Speicherstelle, die durch HL adressiert wird, wird in das angegebene Register geladen.
r kann sein:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Datenfluß:



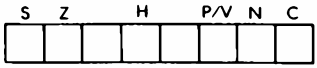
Befehlsablauf: *2 M Zyklen; 7 T Zustände; 3,5 µsek @2 MHz*

Adressierungsart: Indirekt.

Byte Kode:

r:	A	B	C	D	E	H	L
	7E	46	4E	56	5E	66	6E

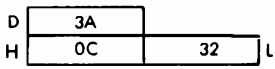
Flags:



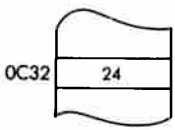
(kein Einfluß)

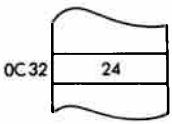
Beispiel: LD D, (HL)

Vorher:



Nachher:





LD SP, HL

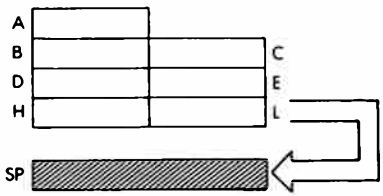
Lade den Stapelzeiger aus HL.

Funktion: SP ← HL

Format:

Beschreibung: Der Inhalt des Registerpaares HL wird in den Stapelzeiger geladen.

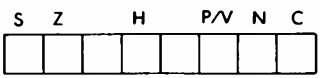
Datenfluß:



Befehlsablauf: *1 M Zyklen; 6 T Zustände; 3 µsek @2 MHz*

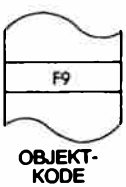
Adressierungsart: Implizit.

Flags:

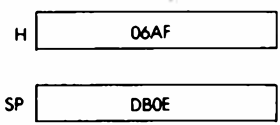


(kein Einfluß)

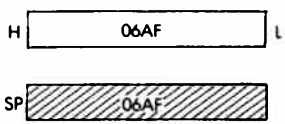
Beispiel: LD SP, HL



Vorher:



Nachher:

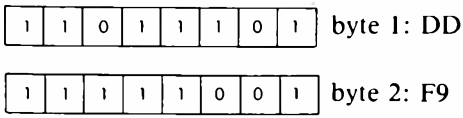


LD SP, IX

Lade den Stapelzeiger aus dem Register IX.

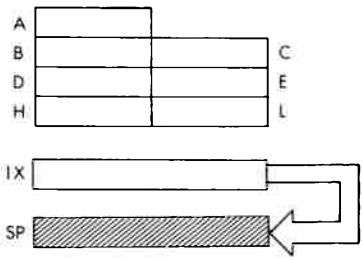
Funktion: SP ← IX

Format:



Beschreibung: Der Inhalt des Registers IX wird in den Stapelzeiger geladen.

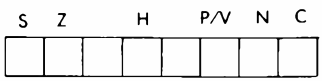
Datenfluß:



Befehlsablauf: *2 M Zyklen; 10 T Zustände; 5 µsek @2 MHz*

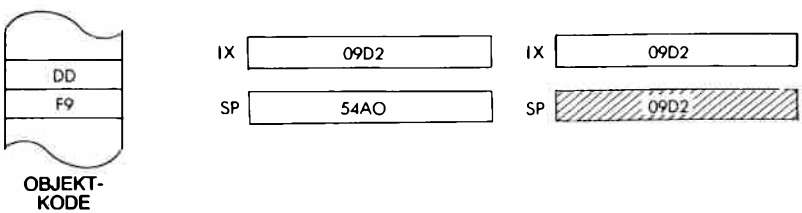
Adressierungsart: Implizit.

Flags:



(kein Einfluß)

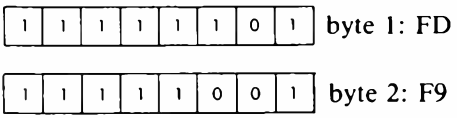
Beispiel: LD SP, IX



LD SP, IY Lade den Stapelzeiger aus dem Register IY.

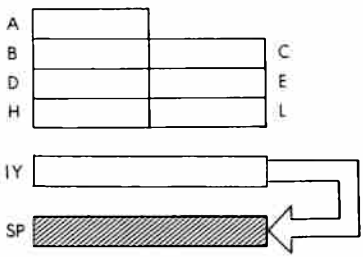
Funktion: SP ← IY

Format:



Beschreibung: Der Inhalt des Registers IY wird in den Stapelzeiger geladen.

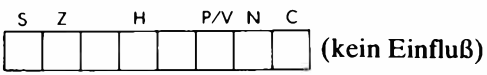
Datenfluß:



Befehlsablauf: *2 M Zyklen; 10 T Zustände; 5 µsek @ 2 MHz*

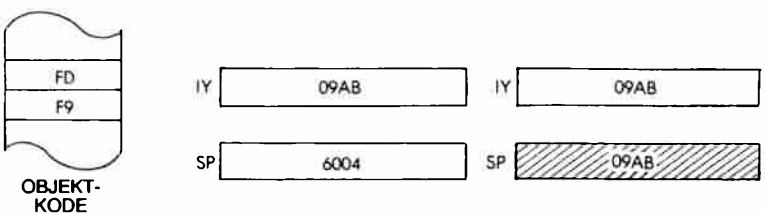
Adressierungsart: Implizit.

Flags:



Beispiel: LD SP, IY

Vorher: Nachher:



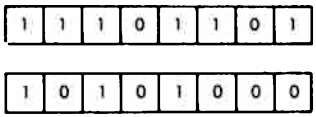
LDD

Blockladen mit Dekrementieren.

Funktion:

(DE) ← (HL); DE ← DE − 1; HL ← HL − 1; BC ← BC − 1

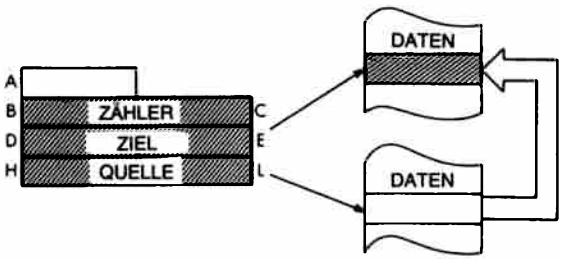
Format:



Beschreibung:

Der Inhalt der Speicherstelle, die durch HL adressiert wird, wird in die Speicherstelle geladen, auf die DE zeigt. Dann werden BC, DE und HL dekrementiert.

Datenfluß:



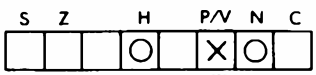
Befehlsablauf:

4 M Zyklen; 16 T Zustände; 8 µsek @ 2 MHz

Adressierungsart:

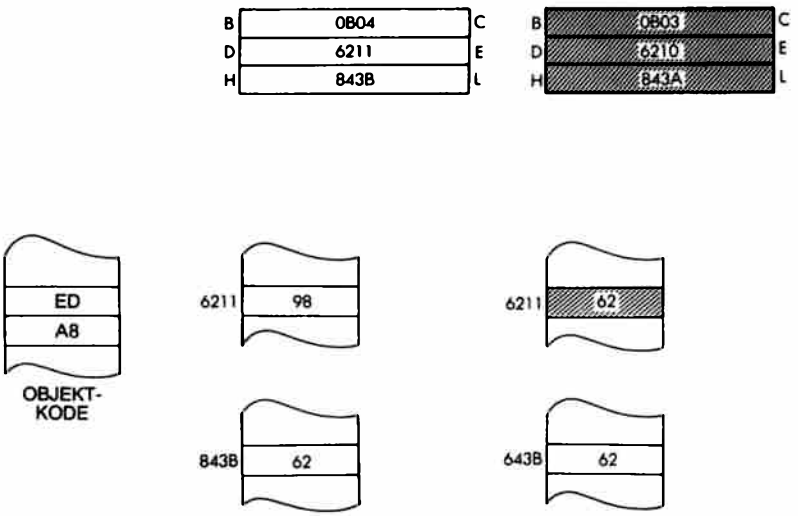
Indirekt.

Flags:



Zurückgesetzt, wenn BC = 0 nach der Ausführung, sonst gesetzt.

Beispiel: LDD



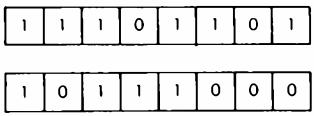
LDDR

Wiederholtes Blockladen mit Dekrementieren.

Funktion:

(DE)←(HL); DE←DE − 1; HL←HL −1;
BC←BC − 1; Wiederhole bis BC = 0

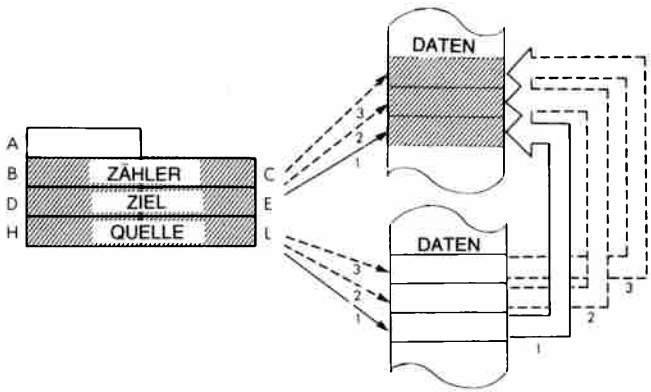
Format:



Beschreibung:

Der Inhalt der Speicherstelle, die durch HL adressiert wird, wird in die Speicherstelle geladen, auf die DE zeigt. Dann werden DE, HL und BC dekrementiert. Ist BC ≠ 0, dann wird der Befehlszähler um zwei dekrementiert und der Befehl wiederholt.

Datenfluß:



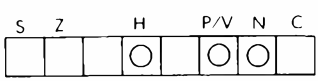
Befehlsablauf:

BC ≠ 0: 5 M Zyklen; 21 T Zustände; 10,5 µsek @ 2 MHz
BC = 0:4 M Zyklen; 16 T Zustände; 8 µsek @ MHz

Adressierungsart:

Indirekt.

Flags:

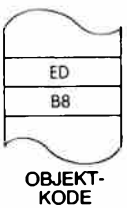


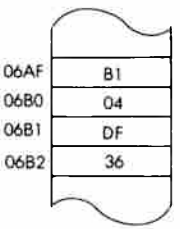
Beispiel: LDDR

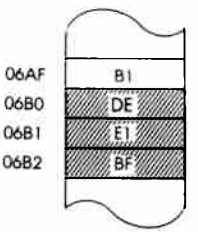
Vorher:

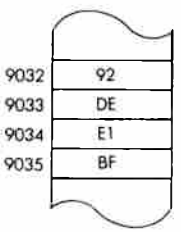
B	0003	C	B	0000	C
D	06B2	E	D	06AF	E
H	9035	L	H	9032	L

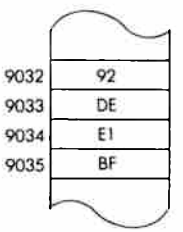
Nachher:











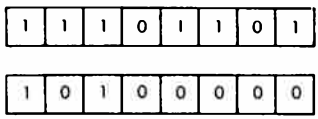
LDI

Blockladen mit Inkrementieren.

Funktion:

(DE) ← (HL); DE ← DE + 1; HL ← HL + 1;
BC ← BC − 1

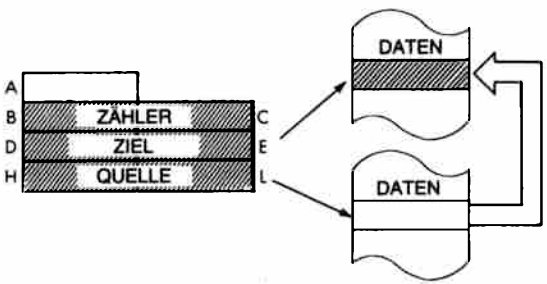
Format:



Beschreibung:

Der Inhalt der Speicherstelle, die durch HL adressiert wird, wird in die Speicherstelle geladen, auf die DE zeigt. Dann werden DE und HL inkrementiert und BC dekrementiert.

Datenfluß:



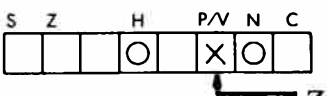
Befehlsablauf:

4 M Zyklen; 16 T Zustände; 8 µsek @ 2 MHz

Adressierungsart:

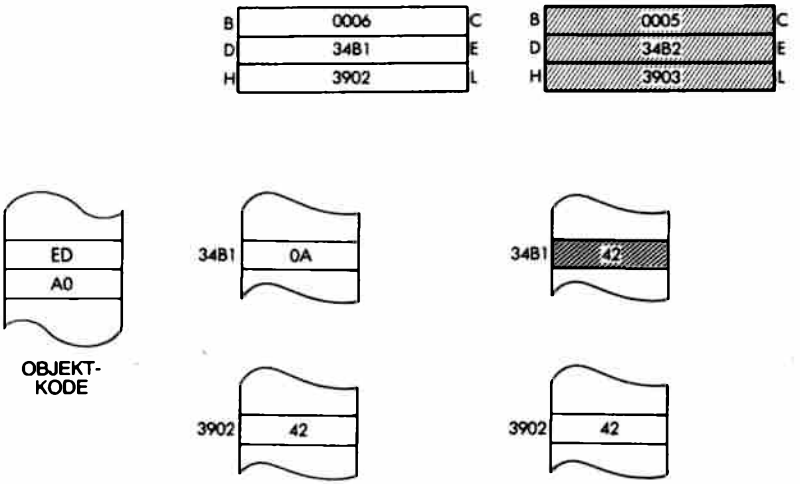
Indirekt.

Flags:



Zurückgesetzt, wenn BC = 0 nach der Ausführung, sonst gesetzt.

Beispiel: **LDI**



LDIR Wiederholtes Blockladen mit Inkrementieren.

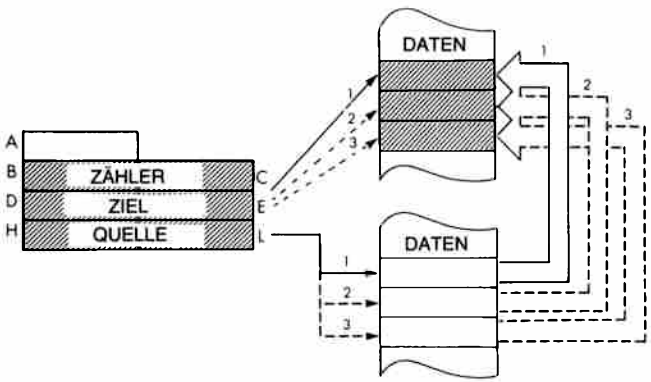
Funktion: (DE)←(HL); DE←DE + 1; HL←HL + 1; BC←BC − 1; Wiederhole bis BC = 0

Format:

1	1	1	0	1	1	0	1	Byte 1: ED
1	0	1	1	0	0	0	0	Byte 2: BO

Beschreibung: Der Inhalt der Speicherstelle, die durch HL adressiert wird, wird in die Speicherstelle geladen, auf die DE zeigt. Dann werden DE und HL inkrementiert und BC dekrementiert. Ist BC ≠ 0, dann wird der Befehlszähler um zwei dekrementiert und der Befehl wiederholt.

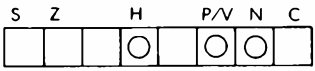
Datenfluß:



Befehlsablauf: *Für BC ≠ 0: 5 M Zyklen; 21 T Zustände; 10,5 µsek @ 2 MHz*
Für BC = 0:4 M Zyklen; 16 T Zustände; 8 µsek @ MHz

Adressierungsart: Indirekt.

Flags:



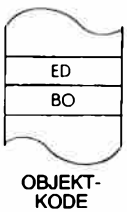
Beispiel: LDIR

Vorher:

B	0002	C
D	4A03	E
H	962A	L

Nachher:

B	0000	C
D	4A05	E
H	962C	L



4A03	12
4A04	F4
4A05	AA

4A03	3B
4A04	90
4A05	AA

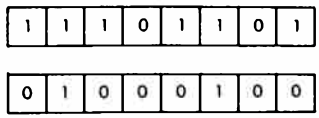
962A	3B
962B	90
962C	6E

962A	3B
962B	90
962C	6E

NEG Negiere Akkumulator.

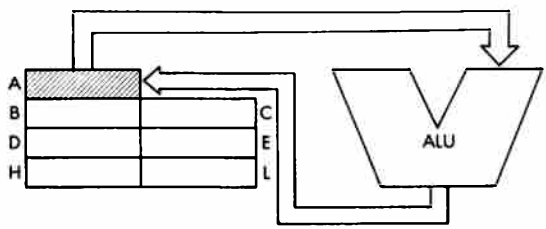
Funktion: A ← 0 − A

Format:



Beschreibung: Der Inhalt des Akkumulators wird von Null subtrahiert (Zweierkomplement). Das Ergebnis wird wieder im Akkumulator gespeichert.

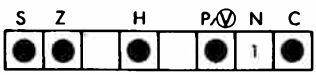
Datenfluß:



Befehlsablauf: *2 M Zyklen; 8 T Zustände; 4 µsek @ 2 MHz*

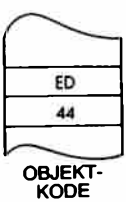
Adressierungsart: Implizit.

Flags:

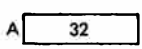


C wird gesetzt, wenn A vor dem Befehl Null war.
P wird gesetzt, wenn A 80H war.

Beispiel: NEG



Vorher:



Nachher:

NOP Keine Operation.

Funktion: Delay.

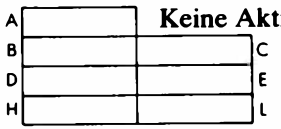
Format:

0	0	0	0	0	0	0	0

00

Beschreibung: Einen M-Zyklus lang wird nichts getan.

Datenfluß:



Befehlsablauf: *1 M Zyklus; 4 T Zustände; 2 µsek @ 2 MHz*

Adressierungsart: Implizit.

Flags:

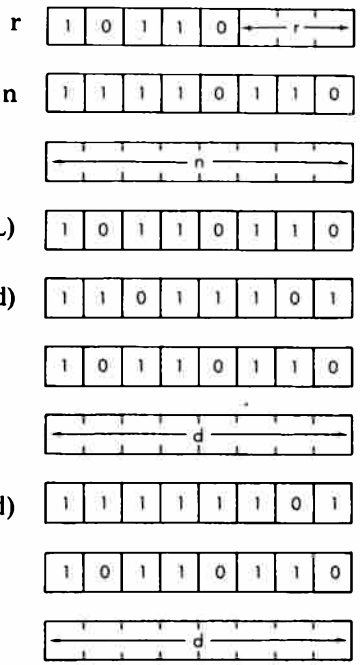
S	Z		H		P/V	N	C

(kein Einfluß)

OR s Logische Oder-Verknüpfung von Akkumulator und Operand s.

Funktion: A ← A V s

Format: *s: kann sein r, n, (HL), (IX + d), oder (IY + d)*

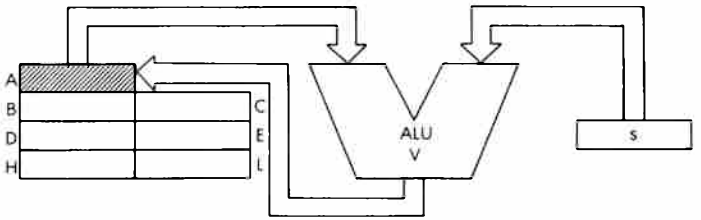


r kann sein:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Beschreibung: Der Akkumulator und der angegebene Operand werden logisch durch die Funktion ODER verknüpft und das Ergebnis wieder im Akkumulator abgelegt. s ist in der Beschreibung des Befehls AND definiert.

Datenfluß:

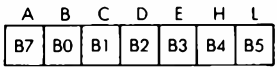


Befehlsablauf:

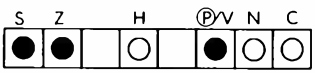
s:	*M Zyklen*	*T Zustände*	*µsek @ 2 MHz:*
r	1	4	4
n	2	7	3.5
(HL)	2	7	3.5
(IX + d)	5	19	9.5
(IY + d)	5	19	9.5

Adressierungsart: r: implizit; n: unmittelbar; (HL): indirekt; (IX + d),(IY + d): indiziert.

Byte Kode: OR r

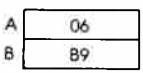


Flags:

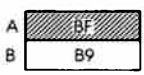


Beispiel: OR B

Vorher:



Nachher:



OTDR

Blockausgabe mit Dekrementieren.

Funktion: (C) ← (HL); B ← B − 1; HL ← HL − 1; Wiederhole bis B = 0.

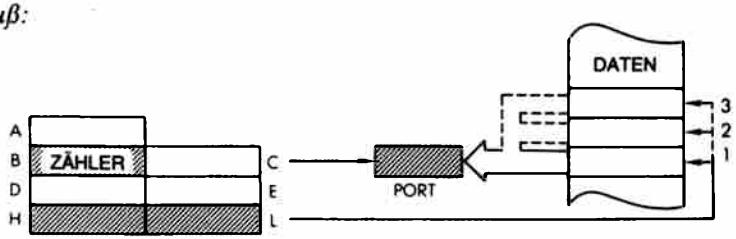
Format:

1	1	1	0	1	1	0	1	Byte 1: ED

1	0	1	1	1	0	1	1	Byte 2: BB

Beschreibung: Der Inhalt der Speicherstelle, die durch das Registerpaar HL adressiert wird, wird zu dem peripheren Gerät ausgegeben, das durch den Inhalt des Registers C adressiert wird. Dann werden die Register B und HL dekrementiert. Ist B ≠0, dann wird der Befehlszähler um zwei dekrementiert und der Befehl nochmals ausgeführt. C liefert die Bits A0 bis A7 des Adreßbusses, B liefert A8 bis A15 (nach dem Dekrementieren).

Datenfluß:



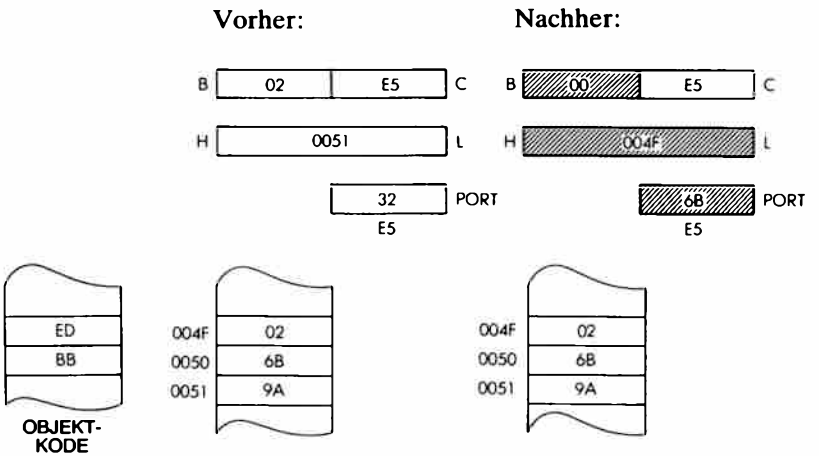
Befehlsablauf: *B = 0: 4 M Zyklen; 16 T Zustände; 8 µsek @ 2 MHz*
C ≠ 0: 5 M Zyklen; 21 T Zustände; 10,5 µsek @ MHz

Adressierungsart: Extern.

Flags:

S	Z		H		P/V	N	C
?	1		?		?	1	

Beispiel: OTDR



OTIR

Blockausgabe mit Inkrementieren.

Funktion:

(C) ← (HL); B ← B − 1; HL ← HL + 1;
Wiederhole bis B = 0.

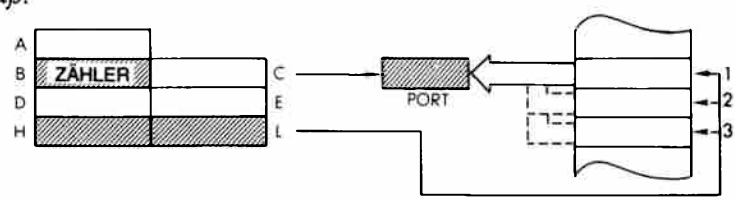
Format:

1	1	1	0	1	1	0	1	Byte 1: ED
1	0	1	1	0	0	1	1	Byte 2: B3

Beschreibung:

Der Inhalt der Speicherstelle, die durch das Registerpaar HL adressiert wird, wird zu dem peripheren Gerät ausgegeben, das durch den Inhalt des Registers C adressiert wird. Dann wird das Register B dekrementiert und HL inkrementiert.
Ist B ≠0, dann wird der Befehlszähler um zwei dekrementiert und der Befehl nochmals ausgeführt.
C liefert die Bits A0 bis A7 des Adreßbusses, B liefert A8 bis A15 (nach dem Dekrementieren).

Datenfluß:



Befehlsablauf:

B = 0: 4 M Zyklen; 16 T Zustände;
8 µsek @ 2 MHz
B ≠ 0: 5 M Zyklen; 21 T Zustände;
10,5 µsek @ MHz

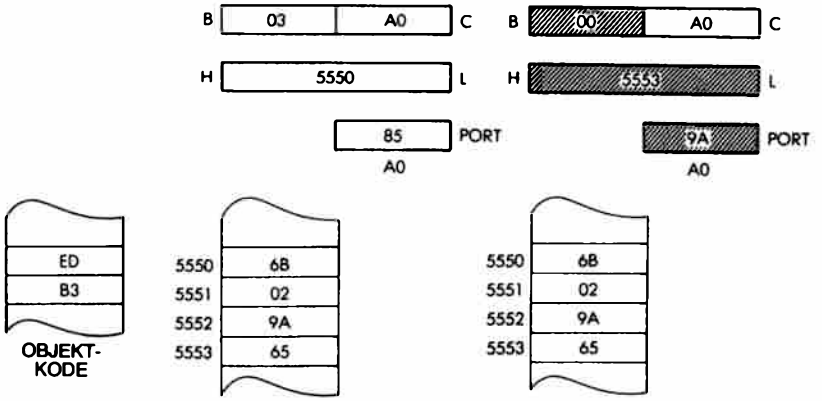
Adressierungsart:

Extern.

Flags:

S	Z		H		P/V	N	C
?	1		?		?	1	

Beispiel: **OTIR**



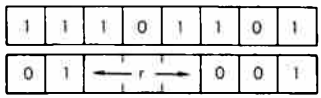
OUT (C), r

Ausgabe des Registers r zum Port C.

Funktion:

(C) ← r

Format:



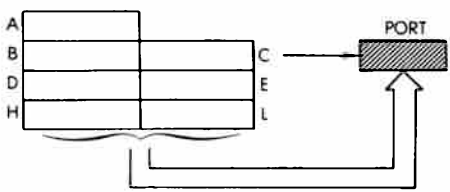
Beschreibung:

Der Inhalt des angegebenen Registers wird an das periphere Gerät ausgegeben, das durch den Inhalt des Registers C adressiert wird.

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

C liefert die Bits A0 bis A7 des Adreßbusses, B liefert A8 bis A15 (nach dem Dekrementieren).

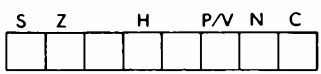
Datenfluß:



Befehlsablauf: *3 M Zyklen; 12 T Zustände; 6 µsek @ 2 MHz*

Adressierungsart: Extern.

Flags:

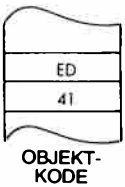


(kein Einfluß)

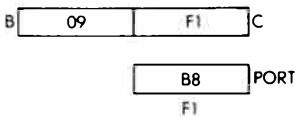
Byte Kode:

r:	A	B	C	D	E	H	L
	79	41	49	51	59	61	69

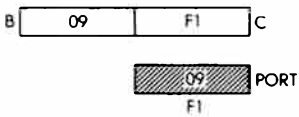
Beispiel: OUT (C), B



Vorher:



Nachher:

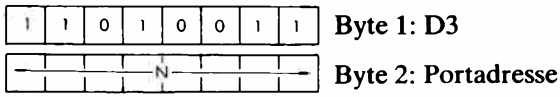


OUT (N), A

Ausgabe des Akkumulators an den peripheren Port N.

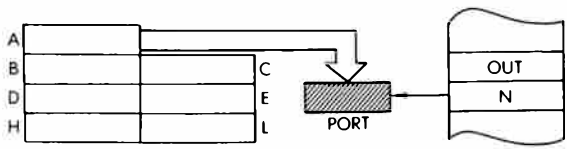
Funktion: (N) ← A

Format:



Beschreibung: Der Inhalt des Akkumulators wird an das periphere Gerät ausgegeben, das durch den Inhalt der Speicherzelle adressiert wird, die unmittelbar auf den Opcode folgt.

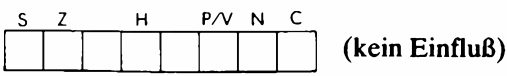
Datenfluß:



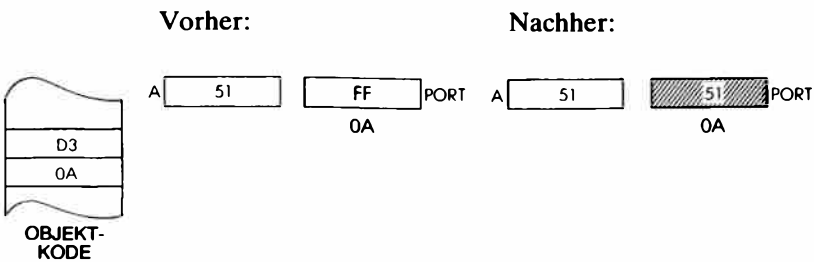
Befehlsablauf: *3 M Zyklen; 11 T Zustände; 5,5 µsek @ 2 MHz*

Adressierungsart: Extern.

Flags:



Beispiel: OUT (0A), A



OUTD

Ausgabe mit Dekrementieren.

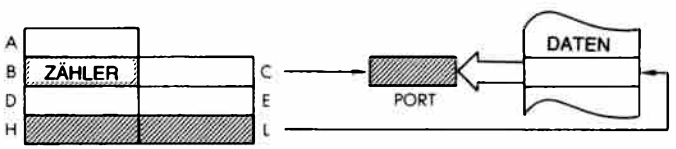
Funktion: (C) ← (HL); B ← B − 1; HL ← HL − 1

Format:

1	1	1	0	1	1	0	1	Byte 1: ED
1	0	1	0	1	0	1	1	Byte 2: AB

Beschreibung: Der Inhalt der Speicherzelle, die durch das Registerpaar HL adressiert wird, wird an das periphere Gerät ausgegeben, das durch den Inhalt des Registers C adressiert wird. Dann werden die Register B und HL dekrementiert. C liefert die Bits A0 bis A7 des Adreßbusses, B liefert A8 bis A15 (nach dem Dekrementieren).

Datenfluß:



Befehlsablauf: *4 M Zyklen; 16 T Zustände; 8 µsek @ 2 MHz*

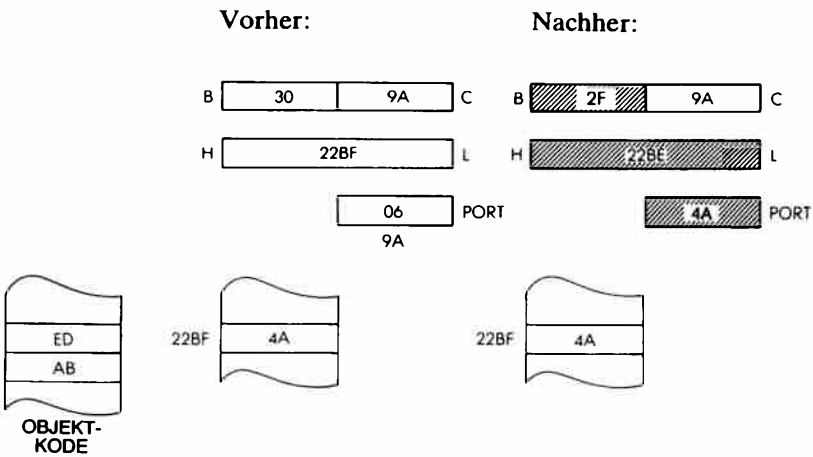
Adressierungsart: Extern.

Flags:

S	Z		H		P/V	N	C
?	X		?		?	1	

Gesetzt, wenn B=0 nach der Ausführung, sonst zurückgesetzt.

Beispiel: OUTD

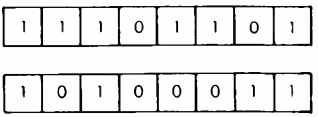


OUTI

Ausgabe mit Inkrementieren.

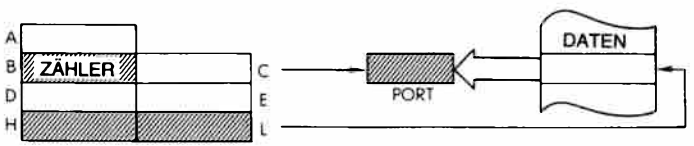
Funktion: (C) ← (HL); B ← B − 1; HL ← HL + 1

Format:



Beschreibung: Der Inhalt der Speicherzelle, die durch das Registerpaar HL adressiert wird, wird an das periphere Gerät ausgegeben, das durch den Inhalt des Registers C adressiert wird. Dann wird das Register B dekrementiert und HL inkrementiert.
C liefert die Bits A0 bis A7 des Adreßbusses, B liefert A8 bis A15 (nach dem Dekrementieren).

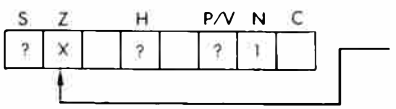
Datenfluß:



Befehlsablauf: *4 M Zyklen; 16 T Zustände; 8 µsek @ 2 MHz*

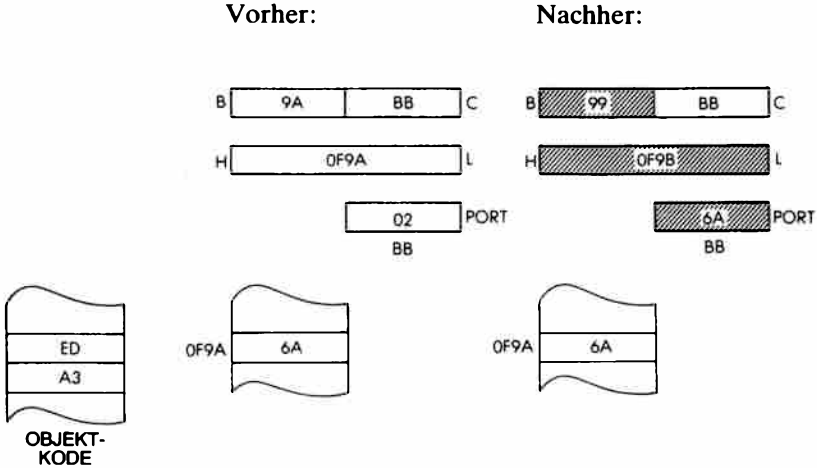
Adressierungsart: Extern.

Flags:



Gesetzt, wenn B=0 nach der Ausführung, sonst zurückgesetzt.

Beispiel: OUTI



POP qq

Hole das Registerpaar qq vom Stapel.

Funktion: $qq_{unten} \leftarrow (SP); qq_{oben} \leftarrow (SP + 1); SP \leftarrow SP + 2$

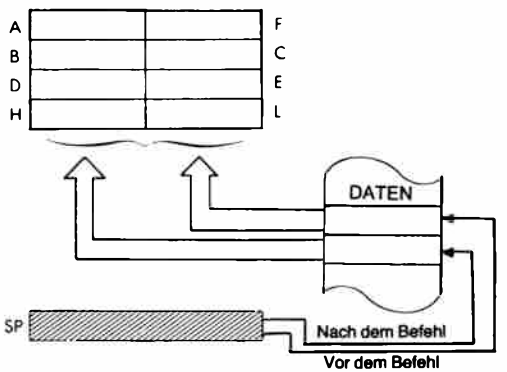
Format:

1	1	q	q	0	0	0	1

Beschreibung: Der Inhalt der Speicherzelle, auf die der Stapelzeiger zeigt, wird in die untere Hälfte des angegebenen Registerpaares geladen. Dann wird der Stapelzeiger inkrementiert und der Inhalt der Speicherzelle, auf die er dann zeigt, in die obere Hälfte des angegebenen Registerpaares geladen. Daraufhin wird der Stapelzeiger nochmals inkrementiert. qq kann sein:

BC – 00	HL – 10
DE – 01	AF – 11

Datenfluß:



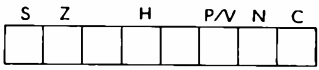
Befehlsablauf: *3 M Zyklen; 10 T Zustände; 5 µsek @ 2 MHz*

Adressierungsart: Indirekt.

Byte Kode:

qq:	BC	DE	HL	AF
	C1	D1	E1	F1

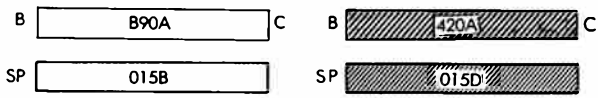
Flags:

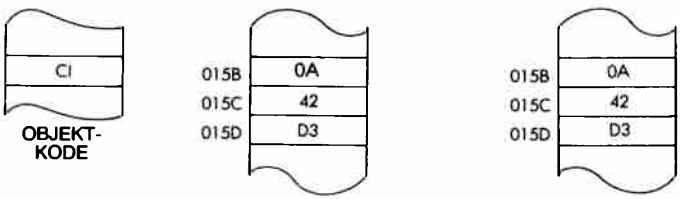


(kein Einfluß)

Beispiel: **POP BC**

Vorher: **Nachher:**





POP IX

Hole das Register IX vom Stapel.

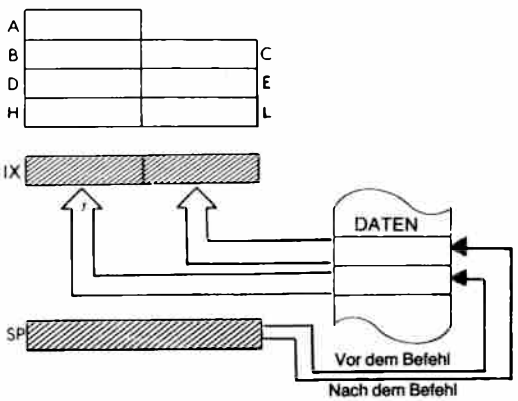
Funktion: $IX_{unten} \leftarrow (SP); IX_{oben} \leftarrow (SP + 1); SP \leftarrow SP + 2$

Format:

1	1	0	1	1	1	0	1	Byte 1: DD
1	1	1	0	0	0	0	1	Byte 2: E1

Beschreibung: Der Inhalt der Speicherzelle, auf die der Stapelzeiger zeigt, wird in die untere Hälfte des Registers IX geladen. Dann wird der Stapelzeiger inkrementiert und der Inhalt der Speicherzelle, auf die er dann zeigt, in die obere Hälfte des Registers IX geladen. Daraufhin wird der Stapelzeiger nochmals inkrementiert.

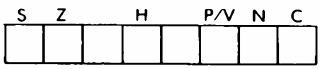
Datenfluß:



Befehlsablauf: *4 M Zyklen; 14 T Zustände; 7 µsek @ 2 MHz*

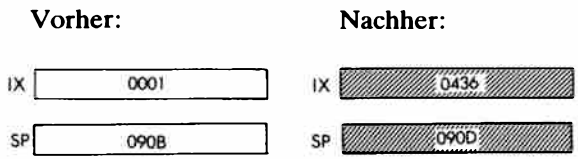
Adressierungsart: Indirekt.

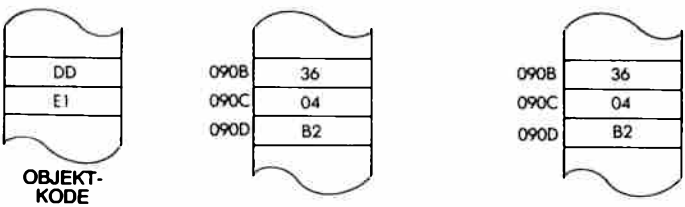
Flags:



(kein Einfluß)

Beispiel: POP IX





POP IY

Hole das Register IY vom Stapel.

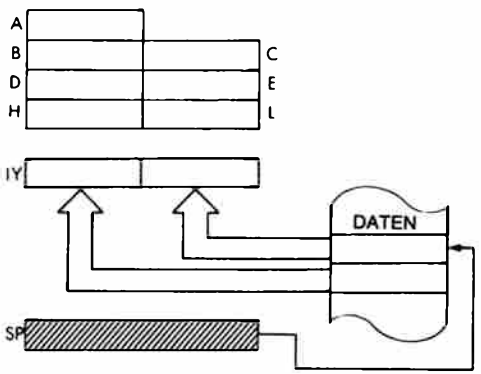
Funktion: $IY_{unten} \leftarrow (SP); IY_{oben} \leftarrow (SP + 1); SP \leftarrow SP + 2$

Format:

1	1	1	1	1	1	0	1	Byte 1: FD
1	1	1	0	0	0	0	1	Byte 2: E1

Beschreibung: Der Inhalt der Speicherzelle, auf die der Stapelzeiger zeigt, wird in die untere Hälfte des Registers IY geladen. Dann wird der Stapelzeiger inkrementiert und der Inhalt der Speicherzelle, auf die er dann zeigt, in die obere Hälfte des Registers IY geladen. Daraufhin wird der Stapelzeiger nochmals inkrementiert.

Datenfluß:



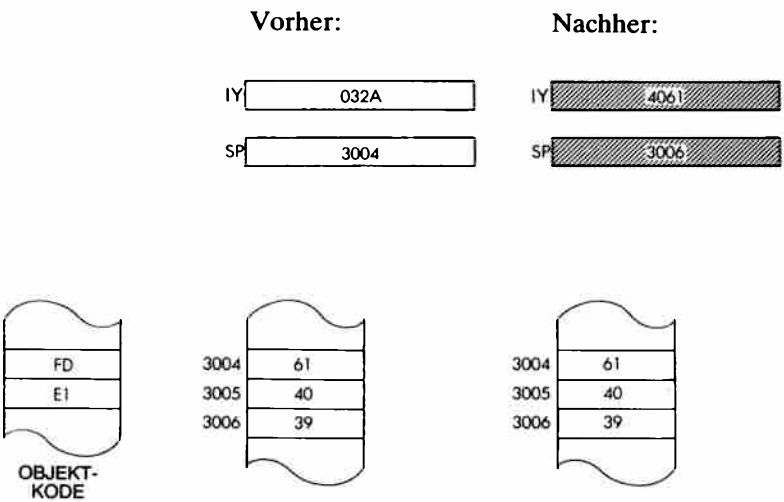
Befehlsablauf: *4 M Zyklen; 14 T Zustände; 2 µsek @ 2 MHz*

Adressierungsart: Indirekt.

Flags:

S	Z		H		P/V	N	C	(kein Einfluß)

Beispiel: POP IY



PUSH qq Lege das Registerpaar qq auf dem Stapel ab.

Funktion: $(SP - 1) \leftarrow qq_{oben}$; $(SP - 2) \leftarrow qq_{unten}$
$SP \leftarrow SP - 2$

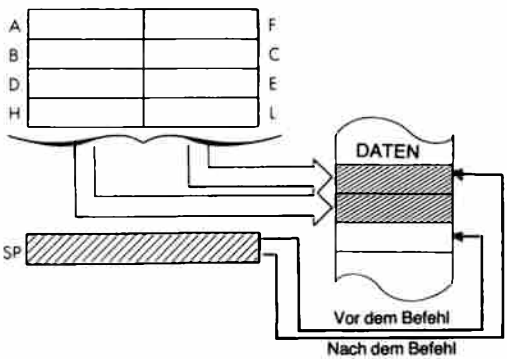
Format:

1	1	q	q	0	1	0	1

Beschreibung: Der Stapelzeiger wird dekrementiert und der Inhalt der oberen Hälfte des angebenen Registerpaares in die Speicherzelle geladen, auf die der Stapelzeiger zeigt. Dann wird der Stapelzeiger nochmals dekrementiert und der Inhalt der unteren Hälfte des Registerpaares in die Speicherzelle geladen, auf die der Stapelzeiger jetzt zeigt.
qq kann sein:

BC - 00	HL - 10
DE - 01	AF - 11

Datenfluß:



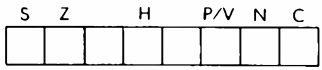
Befehlsablauf: *3 M Zyklen; 11 T Zustände; 6,5 µsek @ 2 MHz*

Adressierungsart: Indirekt.

Byte Kode:

qq:	BC	DE	HL	AF
	C5	D5	E5	F5

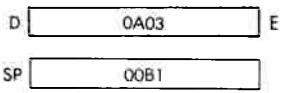
Flags:



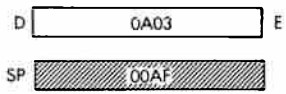
(kein Einfluß)

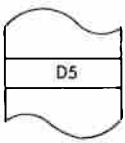
Beispiel: PUSH DE

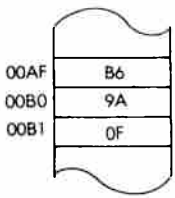
Vorher:

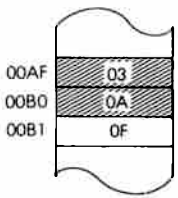


Nachher:









PUSH IX Lege das Register IX auf dem Stapel ab.

Funktion: $(SP - 1) \leftarrow IX_{oben}$; $(SP - 2) \leftarrow IX_{unten}$
$SP \leftarrow SP - 2$

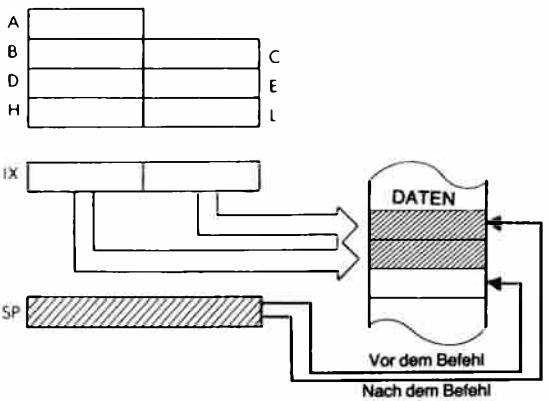
Format:

1	1	0	1	1	1	0	1	Byte 1: DD

1	1	1	0	0	1	0	1	Byte 2: E5

Beschreibung: Der Stapelzeiger wird dekrementiert und der Inhalt der oberen Hälfte des Registers IX in die Speicherzelle geladen, auf die der Stapelzeiger zeigt. Dann wird der Stapelzeiger nochmals dekrementiert und der Inhalt der unteren Hälfte des Registers IX in die Speicherzelle geladen, auf die der Stapelzeiger jetzt zeigt.

Datenfluß:



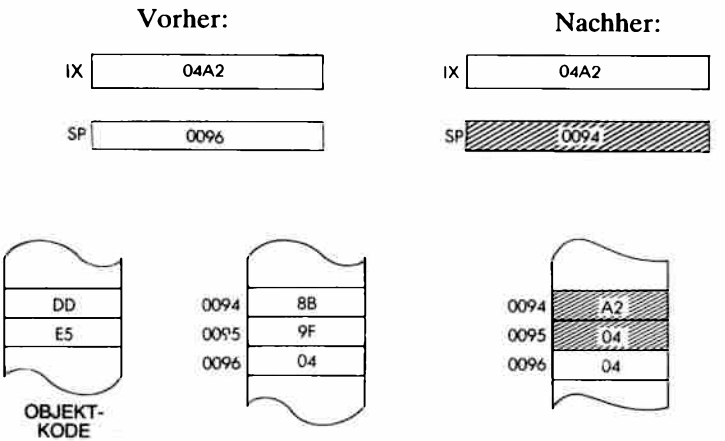
Befehlsablauf: *4 M Zyklen; 15 T Zustände; 7,5 µsek @ 2 MHz*

Adressierungsart: Indirekt.

Flags:

S	Z		H		P/V	N	C	(kein Einfluß)

Beispiel: PUSH IX



PUSH IY Lege das Register IY auf dem Stapel ab.

Funktion: $(SP - 1) \leftarrow IY_{oben}$; $(SP - 2) \leftarrow IY_{unten}$
$SP \leftarrow SP - 2$

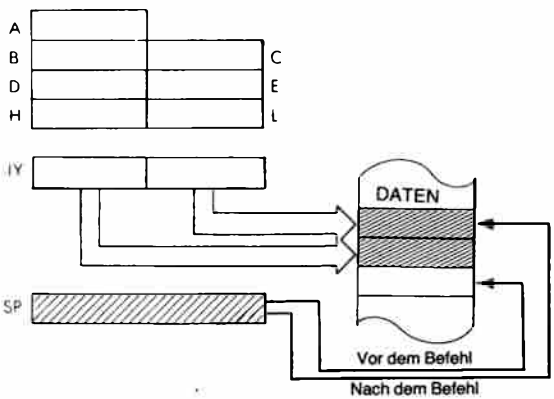
Format:

1	1	1	1	1	1	0	1	Byte 1: FD

1	1	1	0	0	1	0	1	Byte 2: E5

Beschreibung: Der Stapelzeiger wird dekrementiert und der Inhalt der oberen Hälfte des Registers IY in die Speicherzelle geladen, auf die der Stapelzeiger zeigt. Dann wird der Stapelzeiger nochmals dekrementiert und der Inhalt der unteren Hälfte des Registers IY in die Speicherzelle geladen, auf die der Stapelzeiger jetzt zeigt.

Datenfluß:



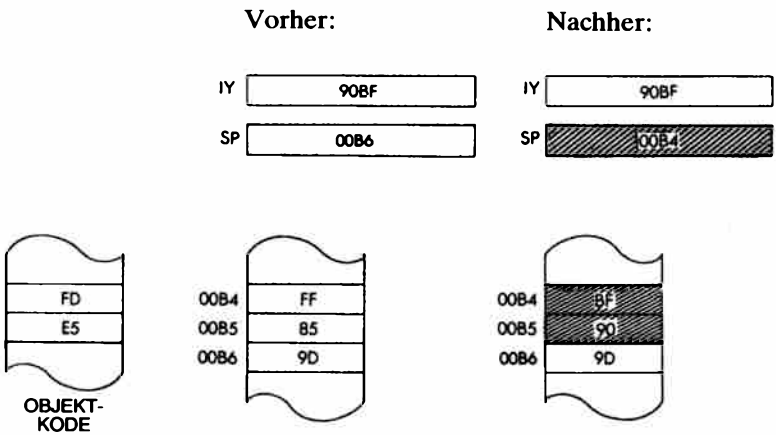
Befehlsablauf: *3 M Zyklen; 15 T Zustände; 7,5 µsek* @ 2 MHz

Adressierungsart: Indirekt.

Flags:

S	Z		H		P/V	N	C	(kein Einfluß)

Beispiel: **PUSH IY**

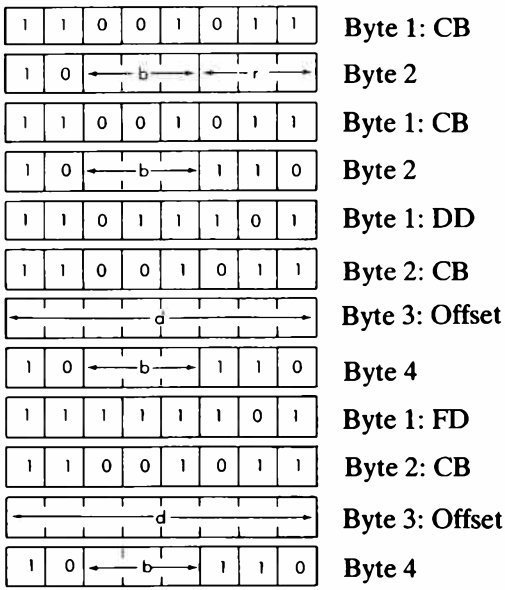


RES b, s Setze Bit b des Operanden s zurück.

Funktion: $s_b \leftarrow 0$

Format:

s:



b kann sein:

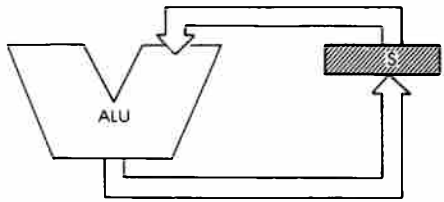
0 – 000	4 – 100
1 – 001	5 – 101
2 – 010	6 – 110
3 – 011	7 – 111

r kann sein:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Beschreibung: Das angegebene Bit in s wird zurückgesetzt. s ist bei der Beschreibung des Befehls BIT definiert.

Datenfluß:



Befehlsablauf:

s:	*M Zyklen*	*T Zustände*	*µsek @ 2 MHz:*
r	2	8	4
(HL)	4	15	7.5
(IX + d)	6	23	11.5
(IY + d)	6	23	11.5

Adressierungsart: r: implizit; (HL): indirekt; (IX d),(IY d): indiziert.

Byte Kode: RES b, r

CB—

b:	r: A	B	C	D	E	H	L
0	87	80	81	82	83	84	85
1	8F	88	89	8A	8B	8C	8D
2	97	90	91	92	93	94	95
3	9F	98	99	9A	9B	9C	9D
4	A7	A0	A1	A2	A0	A4	A5
5	AF	A8	A9	AA	AB	AC	AD
6	B7	B0	B1	B2	B3	B4	B5
7	BF	B8	B9	BA	BB	BC	BD

RES b, (HL)

CB—

b:	0	1	2	3	4	5	6	7
	86	8E	96	9E	A6	AE	B6	BE

RES b, (IX + d) DDCB –
RES b, (HL) CB –
RES b, (IY + d) FDCB –

b:	0	1	2	3	4	5	6	7
	86	8E	96	9E	A6	AE	B6	BE

Flags:

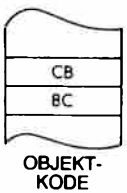
S	Z		H		P/V	N	C

(kein Einfluß)

Beispiel: **RES 1, H**

Vorher: H 42

Nachher: H 40



RET Rücksprung vom Unterprogramm.

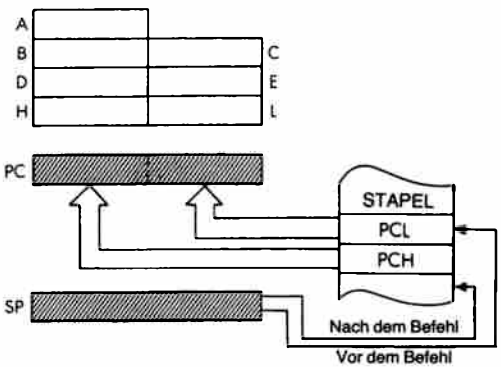
Funktion: $PC_{unten} \leftarrow (SP); PC_{oben} \leftarrow (SP + 1); SP \leftarrow SP + 2$

Format:

1	1	0	0	1	0	0	1	C9

Beschreibung: Der Befehlszähler wird vom Stapel geholt, wie bei dem Befehl POP beschrieben. Der nächste Befehl wird von der Adresse geholt, auf die PC zeigt.

Datenfluß:



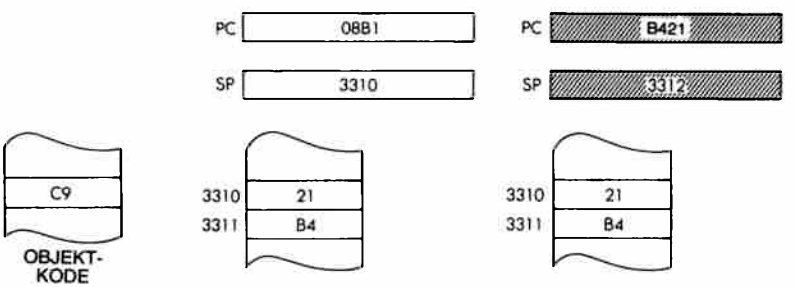
Befehlsablauf: 3 M Zyklen; 10 T Zustände; 5 μsek @ 2 MHz

Adressierungsart: Indirekt.

Flags:

S	Z		H		P/V	N	C	(kein Einfluß)

Beispiel: RET

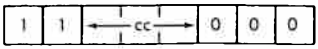


RET cc

Bedingter Rücksprung vom Unterprogramm.

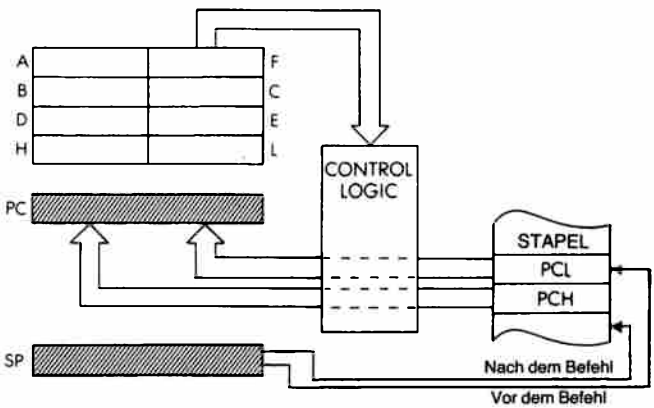
Funktion: Ist cc erfüllt, dann $PC_{unten} \leftarrow (SP); PC_{oben} \leftarrow (SP + 1); SP \leftarrow SP + 2$

Format:



Beschreibung: Ist die Bedingung erfüllt, dann wird der Befehlszähler vom Stapel geholt, wie bei dem Befehl POP beschrieben. Der nächste Befehl wird von der Adresse geholt, auf die PC zeigt. Ist die Bedingung nicht erfüllt, werden die nächsten Befehle der Reihe nach ausgeführt.

Datenfluß:



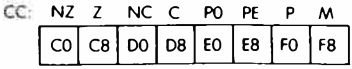
cc kann sein:

NZ – 000	PO – 100
Z – 001	PE – 101
NC – 010	P – 110
C – 011	M – 111

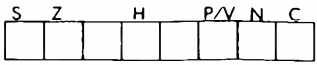
Befehlsablauf: Bedingung erfüllt: 3 M Zyklen; 11 T Zustände; 6,5 µsek @ 2 MHz
Bedingung nicht erfüllt: 1 M Zyklen; 5 T Zustände; 2,5 µsek @ 2 MHz

Adressierungsart: Indirekt.

Byte Kode:



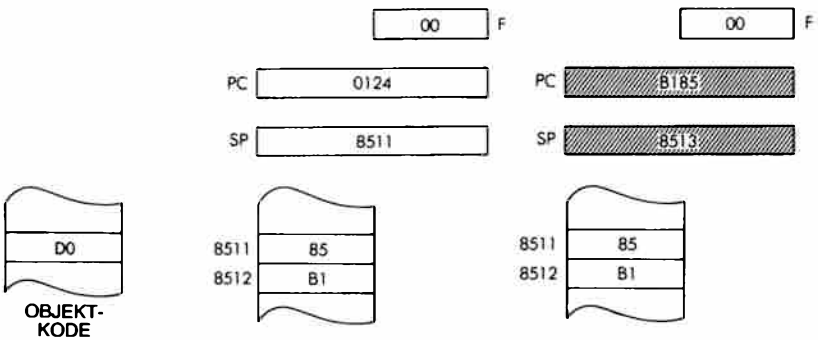
Flags:



(kein Einfluß)

Beispiel: RET NC

Vorher: **Nachher:**



RETI Rücksprung vom Interrupt-Behandlungs-Programm.

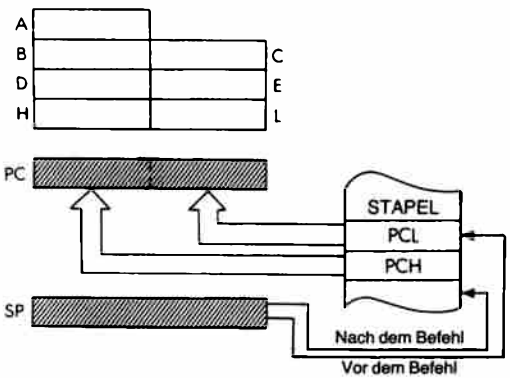
Funktion: $PC_{unten} \leftarrow (SP); PC_{oben} \leftarrow (SP + 1); SP \leftarrow SP + 2$

Format:

1	1	1	0	1	1	0	1	Byte 1: ED
0	1	0	0	1	1	0	1	Byte 2: 4D

Beschreibung: **Der Befehlszähler wird vom Stapel geholt, wie bei dem Befehl POP beschrieben. Die Peripheriebausteine von Zilog erkennen diesen Befehl als das Ende einer Behandlungsroutine, so daß die entsprechende Bearbeitung verschachtelter Interrupts ermöglicht wird. Vor RETI muß ein EI ausgeführt werden, um Interrupts wieder freizugeben.**

Datenfluß:



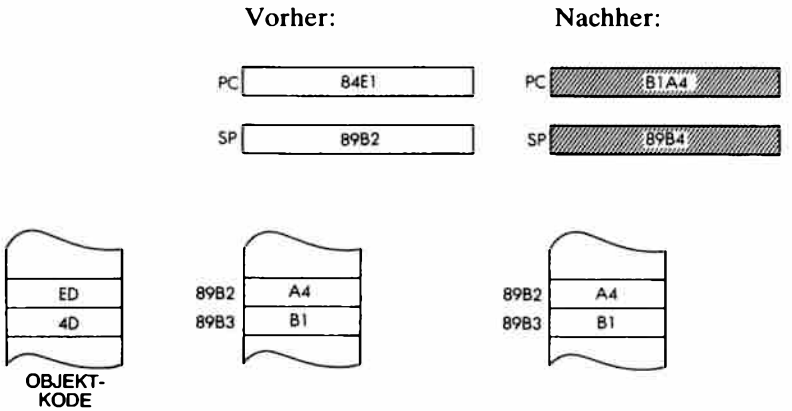
Befehlsablauf: 4 M Zyklen; 14 T Zustände; 7 μsek @ 2 MHz

Adressierungsart: Indirekt.

Flags:

S	Z		H		P/V	N	C	(kein Einfluß)

Beispiel: **RETI**



RETN Rücksprung von einem nicht-maskierbaren Interrupt.

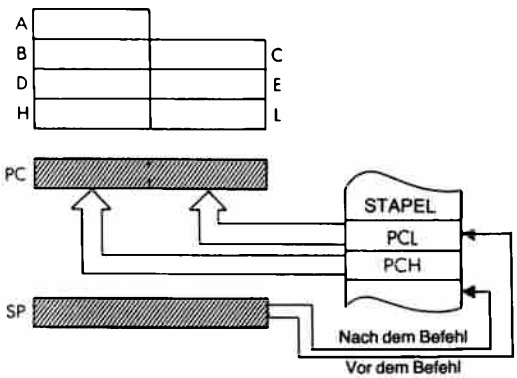
Funktion: $PC_{unten} \leftarrow (SP); PC_{oben} \leftarrow (SP + 1); SP \leftarrow SP + 2;$ IFF1 ← IFF2

Format:

1	1	1	0	1	1	0	1	Byte 1: ED
0	1	0	0	0	1	0	1	Byte 2: 45

Beschreibung: Der Befehlszähler wird vom Stapel geholt, wie bei dem Befehl POP beschrieben. Dann wird der Inhalt von IFF2 (Zwischenspeicher) wieder nach IFF1 kopiert, um den Zustand des Interruptflags vor dem nicht-maskierbaren Interrupt wiederherzustellen.

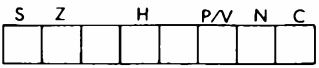
Datenfluß:



Befehlsablauf: 4 M Zyklen; 14 T Zustände; 7 µsek @ 2 MHz

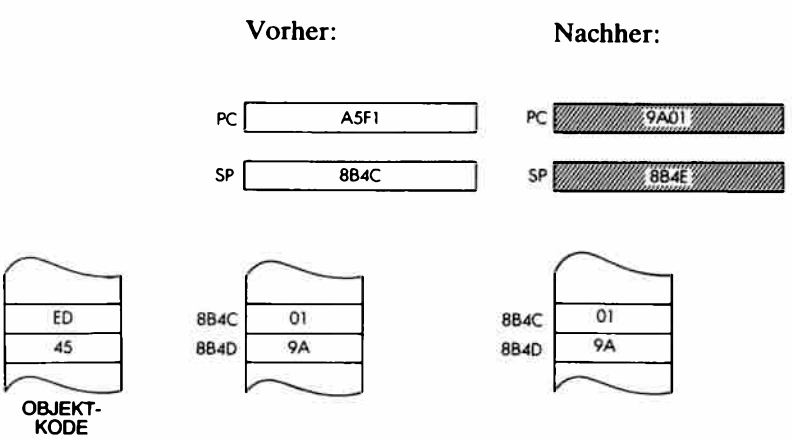
Adressierungsart: Indirekt.

Flags:



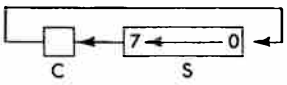
(kein Einfluß)

Beispiel: RETN



RL s Rotiere links durch das Übertragsbit (Carry).

Funktion:



Format: *s:*

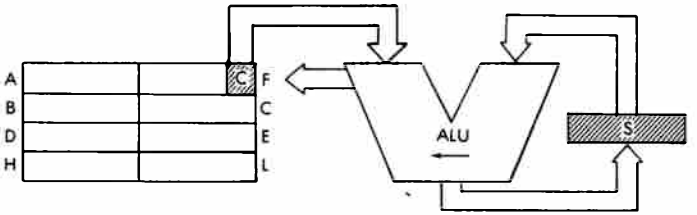
r	1 1 0 0 1 0 1 1	Byte 1: CB
	0 0 0 1 0 r r r	Byte 2
(HL)	1 1 0 0 1 0 1 1	Byte 1: CB
	0 0 0 1 0 1 1 0	Byte 2: 16
(IX + d)	1 1 0 1 1 1 0 1	Byte 1: DD
	1 1 0 0 1 0 1 1	Byte 2: CB
	d d d d d d d d	Byte 3: Offset
	0 0 0 1 0 1 1 0	Byte 4: 16
(IY + d)	1 1 1 1 1 1 0 1	Byte 1: FD
	1 1 0 0 1 0 1 1	Byte 2: CB
	d d d d d d d d	Byte 3: Offset
	0 0 0 1 0 1 1 0	Byte 4: 16

r kann sein:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Beschreibung: Der Inhalt der Stelle, die durch den Operanden festgelegt wird, wird links verschoben. Der Inhalt des Übertragsbits wird nach Bit 0 und der Inhalt von Bit 7 ins Übertragsbit verschoben. Das Ergebnis wird wieder an der alten Stelle abgelegt. s ist bei der Beschreibung des Befehls RLC definiert.

Datenfluß:

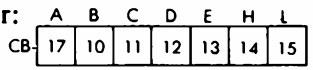


Befehlsablauf:

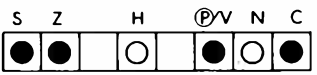
s:	*M Zyklen*	*T Zustände*	*µsek @ 2 MHz:*
r	2	8	4
(HL)	4	15	7.5
(IX + d)	6	23	11.5
(IY + d)	6	23	11.5

Adressierungsart: r: implizit; (HL): indirekt; (IX d),(IY d): indiziert.

Byte Kode: RL r

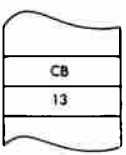


Flags:

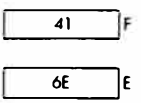


C wird von Bit 7 der Quelle gesetzt.

Beispiel: RL E



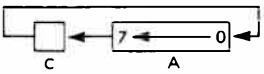
Vorher:



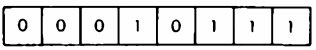
Nachher:

RLA Rotiere den Akkumulator links durchs Übertragsbit.

Funktion:



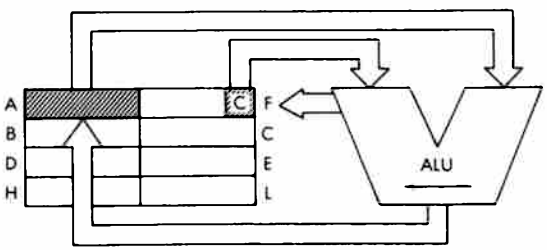
Format:



17

Beschreibung: Der Inhalt des Akkumulators wird nach links verschoben. Der Inhalt des Übertragsbits wird nach Bit 0, der Inhalt von Bit 7 ins Übertragsbit verschoben. (9 Bit Rotation).

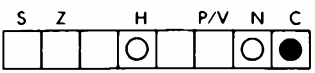
Datenfluß:



Befehlsablauf: 1 M Zyklus; 4 T Zustände; 2 μsek @ 2 MHz

Adressierungsart: Implizit.

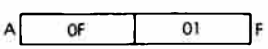
Flags:



C wird gesetzt durch Bit 7 von A.

Beispiel: RLA

Vorher:

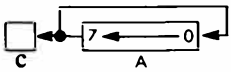


Nachher:

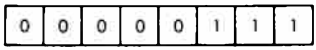
RLCA

Rotiere Akkumulator links.

Funktion:



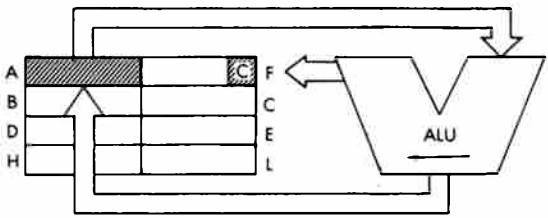
Format:



07

Beschreibung: Der Inhalt des Akkumulators wird um eine Stelle links rotiert. Der ursprüngliche Inhalt von Bit 7 wird ins Übertragsflag und gleichzeitig ins Bit 0 verschoben.

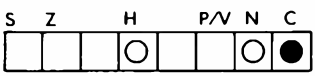
Datenfluß:



Befehlsablauf: 1 M Zyklus; 4 T Zustände; 2 μsek @ 2 MHz

Adressierungsart: Implizit.

Flags:



C wird Bit 7 in A gesetzt.

Beispiel: RLCA

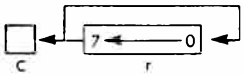
Vorher: Nachher:

Achtung:
Mit Ausnahme der Flags ist dieser Befehl identisch zu RLC A. Er wurde wegen der Kompatibilität zum 8080 eingebaut.

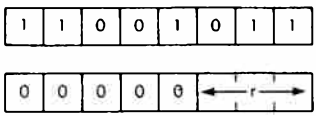
RLC r

Rotiere links.

Funktion:



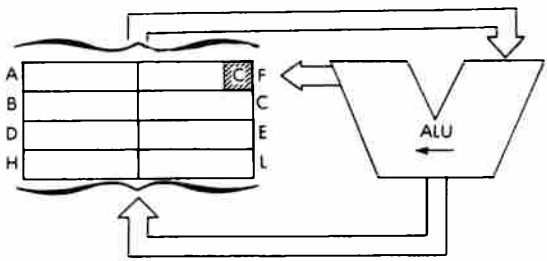
Format:



Beschreibung: Der Inhalt der durch den Operanden bestimmten Stelle wird links rotiert und das Ergebnis an der alten Stelle abgelegt. Der Inhalt von Bit 7 wird ins Übertragsflag und gleichzeitig ins Bit 0 verschoben. r kann sein:

B – 000	H – 100
C – 001	L – 101
D – 010	

Datenfluß:



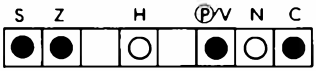
Befehlsablauf: 2 M Zyklen; 8 T Zustände; 4 μsek @ 2 MHz

Adressierungsart: Implizit.

Byte Kode: r:

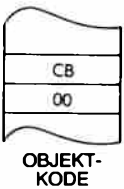
	A	B	C	D	E	H	L
CB-	07	00	01	02	03	04	05

Flags:

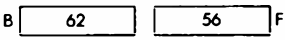


C wird von Bit 7 der Quelle gesetzt.

Beispiel: RLC B



Vorher:

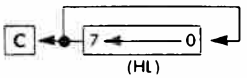


Nachher:

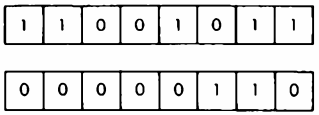
RLC (HL)

Rotiere die Speicherzelle (HL) links.

Funktion:

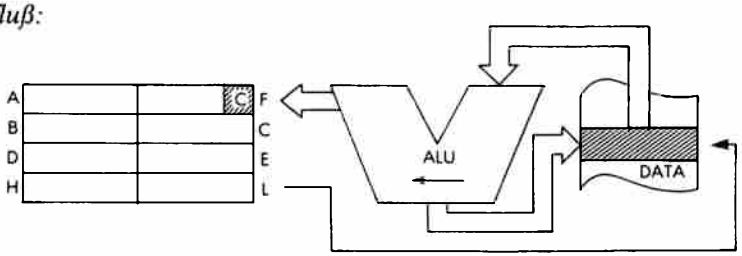


Format:



Beschreibung: Der Inhalt der Speicherzelle, die durch das Registerpaar HL adressiert wird, wird links rotiert und das Ergebnis an der alten Stelle abgelegt. Der Inhalt von Bit 7 wird ins Übertragsflag und gleichzeitig ins Bit 0 verschoben.

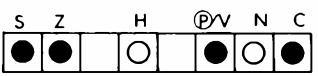
Datenfluß:



Befehlsablauf: 4 M Zyklen; 15 T Zustände; 7,5 µsek @ 2 MHz

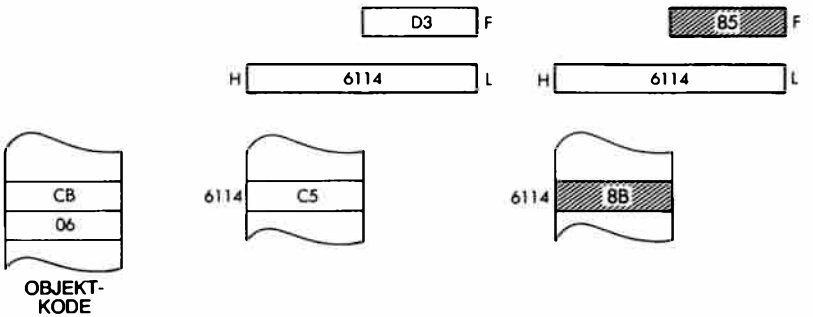
Adressierungsart: Indirekt.

Flags:



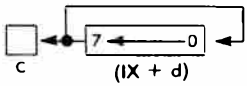
C wird von Bit 7 der Quelle gesetzt.

Beispiel: RLC (HL)

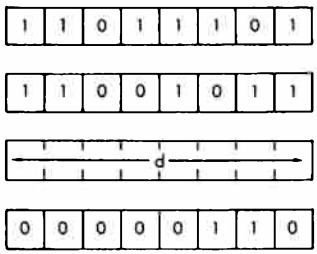


RLC (IX + d) Rotiere die Speicherzelle (IX + d) links.

Funktion:

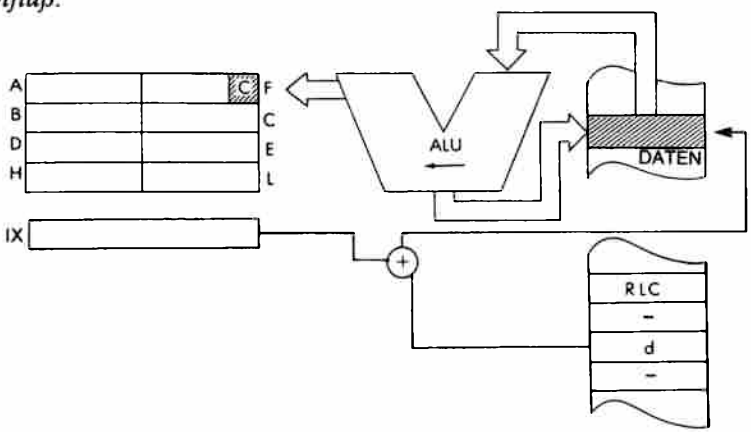


Format:



Beschreibung: Der Inhalt der Speicherzelle, die durch den Inhalt des Registers IX plus einem gegebenen Offset adressiert wird, wird links rotiert und das Ergebnis an der alten Stelle abgelegt. Der Inhalt von Bit 7 wird ins Übertragsflag und gleichzeitig ins Bit 0 verschoben.

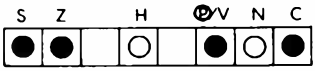
Datenfluß:



Befehlsablauf: 6 M Zyklen; 23 T Zustände; 11,5 μsek @ 2 MHz

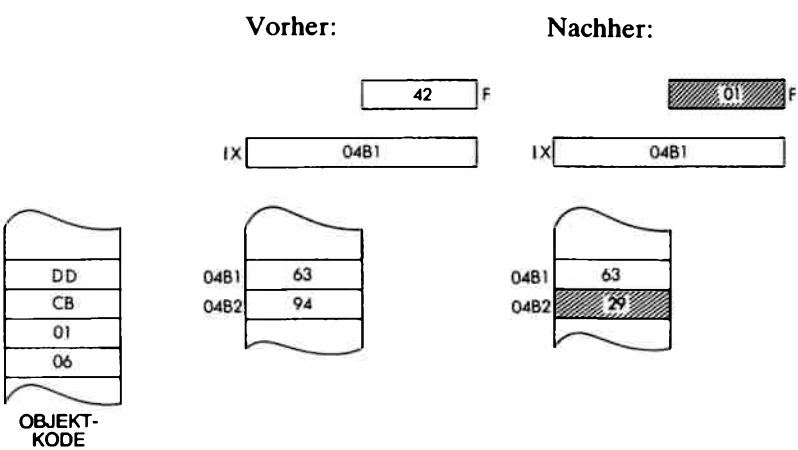
Adressierungsart: Indiziert.

Flags:



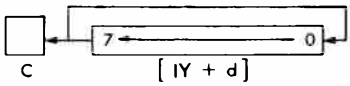
C wird von Bit 7 der Quelle gesetzt.

Beispiel: RLC (IX + 1)

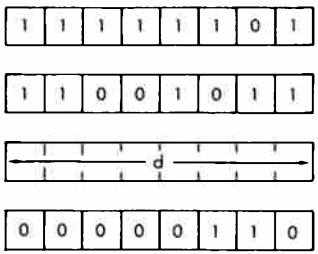


RLC (IY + d) Rotiere die Speicherzelle (IY + d) links.

Funktion:

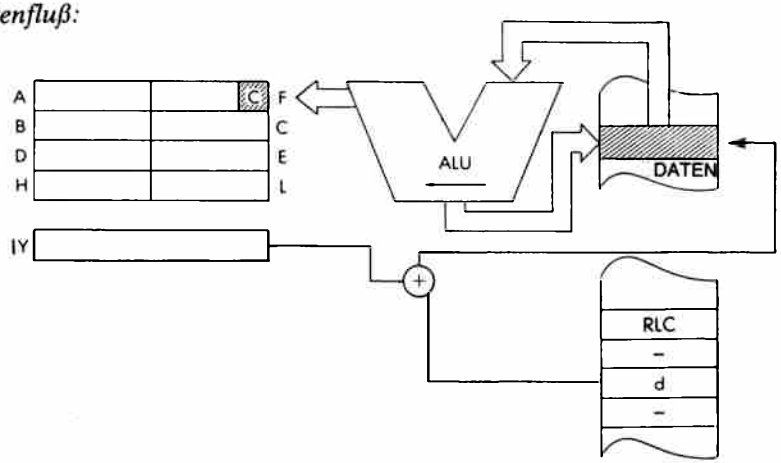


Format:



Beschreibung: Der Inhalt der Speicherzelle, die durch den Inhalt des Registers IY plus einem gegebenen Offset adressiert wird, wird links rotiert und das Ergebnis an der alten Stelle abgelegt. Der Inhalt von Bit 7 wird ins Übertragsflag und gleichzeitig ins Bit 0 verschoben.

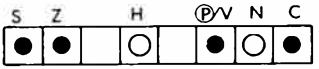
Datenfluß:



Befehlsablauf: 6 M Zyklen; 23 T Zustände; 11,5 μsek @ 2 MHz

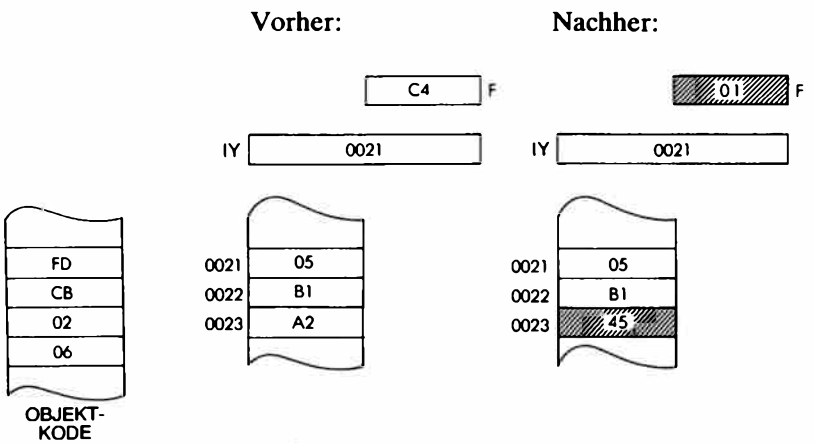
Adressierungsart: Indiziert.

Flags:



C wird von Bit 7 der Quelle gesetzt.

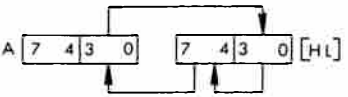
Beispiel: RLC (IY + 2)



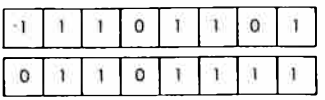
RLD

Rotiere links dezimal.

Funktion:



Format:



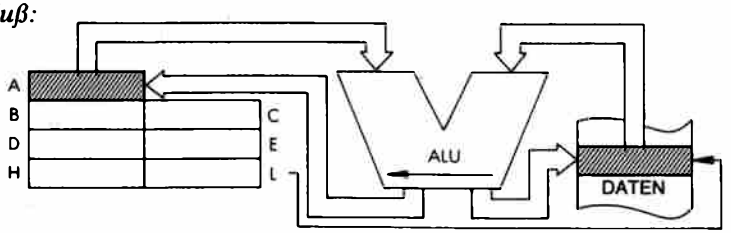
Byte 1: ED

Byte 2: 6F

Beschreibung:

Die vier unteren Bit der Speicherzelle, die durch den Inhalt von HL adressiert wird, werden in die oberen Bits der gleichen Stelle verschoben. Die vier oberen Bit kommen in die vier unteren Bit des Akkumulators. Die unteren Bit des Akkumulators kommen in die vier unteren Bit der ursprünglich festgelegten Speicherzelle. Alle diese Operationen werden gleichzeitig ausgeführt.

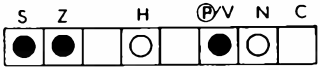
Datenfluß:



Befehlsablauf: 5 M Zyklen; 18 T Zustände; 9 μsek @ 2 MHz

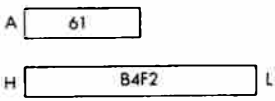
Adressierungsart: Indirekt.

Flags:

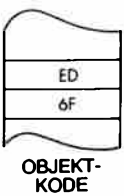


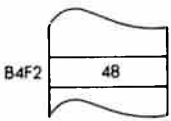
Beispiel: RLD

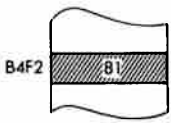
Vorher:



Nachher:

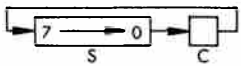






RR s Rotiere rechts durch das Übertragsbit C.

Funktion:



Format:

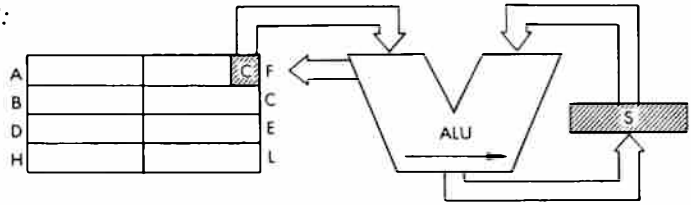
r	1 1 0 0 1 0 1 1	Byte 1: CB
	0 0 0 1 1 r r r	Byte 2
(HL)	1 1 0 0 1 0 1 1	Byte 1: CB
	0 0 0 1 1 1 1 0	Byte 2: 1E
(IX + d)	1 1 0 1 1 1 0 1	Byte 1: DD
	1 1 0 0 1 0 1 1	Byte 2: CB
	d d d d d d d d	Byte 3: Offset
	0 0 0 1 1 1 1 0	Byte 4: 1E
(IY + d)	1 1 1 1 1 1 0 1	Byte 1: FD
	1 1 0 0 1 0 1 1	Byte 2: CB
	d d d d d d d d	Byte 3: Offset
	0 0 0 1 1 1 1 0	Byte 4: 1E

r kann sein:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Beschreibung: Der Inhalt der Stelle, die durch den Operanden festgelegt wird, wird rechts verschoben. Der Inhalt des Übertragsbits wird nach Bit 7 und der Inhalt von Bit 0 ins Übertragsbit verschoben. Das Ergebnis wird wieder an der alten Stelle abgelegt. s ist bei der Beschreibung des Befehls RLC definiert.

Datenfluß:



Befehlsablauf:

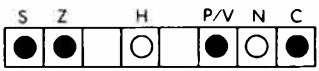
s:	*M Zyklen*	*T Zustände*	*µsek @ 2 MHz*
r	2	8	4
(HL)	4	15	7.5
(IX + d)	6	23	11.5
(IY + d)	6	23	11.5

Adressierungsart: r: implizit; (HL): indirekt; (IX d),(IY d): indiziert.

Byte Kode: RR r:

r:	A	B	C	D	E	H	L
CB-	1F	18	19	1A	1B	1C	1D

Flags:

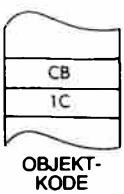


C wird von Bit 0 der Quelle gesetzt.

Beispiel: RR H

Vorher:

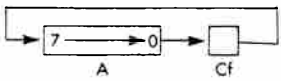
Nachher:



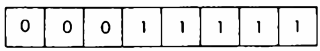
RRA

Rotiere den Akkumulator rechts durchs Übertragsbit.

Funktion:



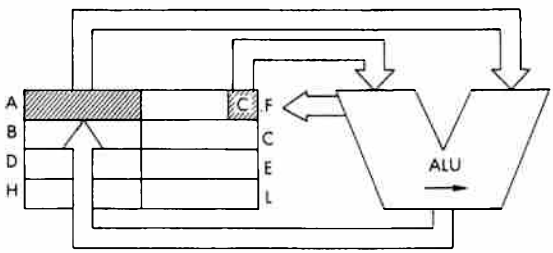
Format:



1F

Beschreibung: Der Inhalt des Akkumulators wird rechts verschoben. Der Inhalt des Übertragsbits wird nach Bit 7, der Inhalt von Bit 0 ins Übertragsbit verschoben (9 Bit Rotation).

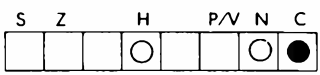
Datenfluß:



Befehlsablauf: 1 M Zyklus; 4 T Zustände; 2 μsek @ 2 MHz

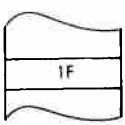
Adressierungsart: Implizit.

Flags:

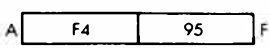


C wird gesetzt durch Bit 0 von A.

Beispiel: RRA



Vorher:

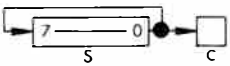


Nachher:

Achtung: Dieser Befehl ist weitgehend identisch mit RR A. Er wurde wegen der Kompatibilität zum 8080 eingebaut.

RRC s Rotiere rechts.

Funktion:



Format: s: *s kann sein r,(HL),(IX + d),(IY + d).*

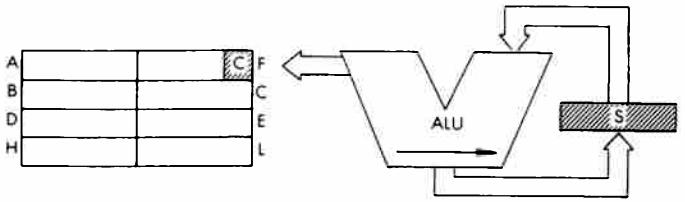
r	1 1 0 0 1 0 1 1	Byte 1: CB
	0 0 0 0 1 r r r	Byte 2
(HL)	1 1 0 0 1 0 1 1	Byte 1: CB
	0 0 0 0 1 1 1 0	Byte 2: 0E
(IX + d)	1 1 0 1 1 1 0 1	Byte 1: DD
	1 1 0 0 1 0 1 1	Byte 2: CB
	d d d d d d d d	Byte 3: Offset
	0 0 0 0 1 1 1 0	Byte 4: 0E
(IY + d)	1 1 1 1 1 1 0 1	Byte 1: FD
	1 1 0 0 1 0 1 1	Byte 2: CB
	d d d d d d d d	Byte 3: Offset
	0 0 0 0 1 1 1 0	Byte 4: 0E

r kann sein:

A – 111	E – 011
B – 000	H – 100
C – 011	L – 101
D – 010	

Beschreibung: Der Inhalt der durch den Operanden bestimmten Stelle wird rechts rotiert und das Ergebnis an der alten Stelle abgelegt. Der Inhalt von Bit 0 wird ins Übertragsflag und gleichzeitig ins Bit 7 verschoben. s ist bei der Beschreibung des Befehls RLC definiert.

Datenfluß:



Befehlsablauf:

s:	*M Zyklen*	*T Zustände*	*µsek @ 2 MHz*
r	2	8	4
(HL)	4	15	7.5
(IX + d)	6	23	11.5
(IY + d)	6	23	11.5

Adressierungsart: r: implizit; (HL): indirekt; (IX d),(IY d): indiziert.

Byte Kode: RRC r

r:	A	B	C	D	E	H	L
CB-	0F	08	09	0A	0B	0C	0D

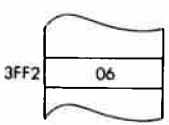
Flags:

S	Z		H		P/V	N	C
●	●		○		●	○	●

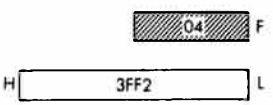
C wird von Bit 0 der Quelle gesetzt.

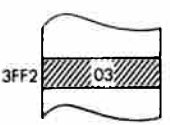
Beispiel: RRC (HL)

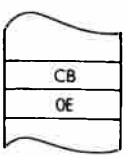
Vorher:



Nachher:



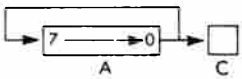




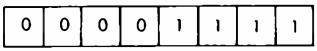
RRCA

Rotiere Akkumulator rechts.

Funktion:

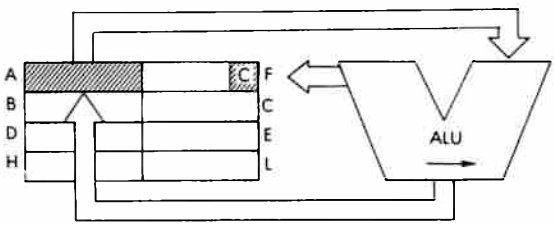


Format:



Beschreibung: Der Inhalt des Akkumulators wird um eine Stelle rechts rotiert. Der ursprüngliche Inhalt von Bit 0 wird ins Übertragsflag und gleichzeitig ins Bit 7 verschoben.

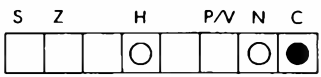
Datenfluß:



Befehlsablauf: 1 M Zyklus; 4 T Zustände; 2 μsek @ 2 MHz

Adressierungsart: Implizit.

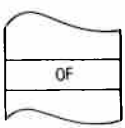
Flags:

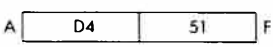


C wird von Bit 0 in A gesetzt.

Beispiel: RRCA

Vorher: Nachher:

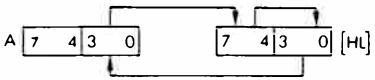




RRD

Rotiere rechts dezimal.

Funktion:



Format:

1	1	1	0	1	1	0	1

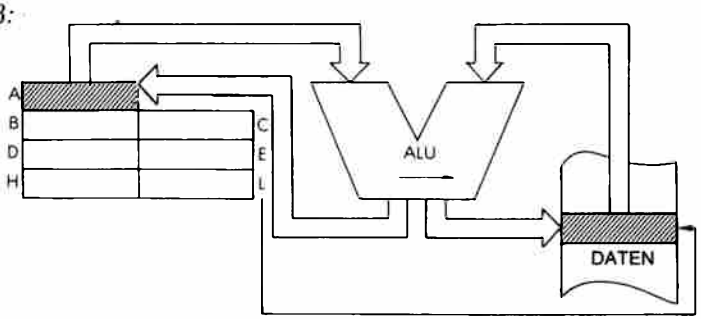
Byte 1: ED

0	1	1	0	0	1	1	1

Byte 2: 67

Beschreibung: Die vier oberen Bit der Speicherzelle, die durch den Inhalt von HL adressiert wird, werden in die unteren Bits der gleichen Stelle verschoben. Die vier unteren Bit kommen in die vier unteren Bit des Akkumulators. Die unteren Bit des Akkumulators kommen in die vier oberen Bit der ursprünglich festgelegten Speicherzelle. Alle diese Operationen werden gleichzeitig ausgeführt.

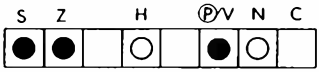
Datenfluß:



Befehlsablauf: 5 M Zyklen; 18 T Zustände; 9 μsek @ 2 MHz

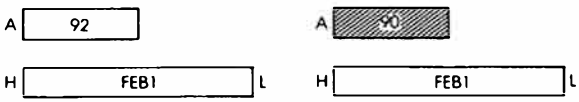
Adressierungsart: Indirekt.

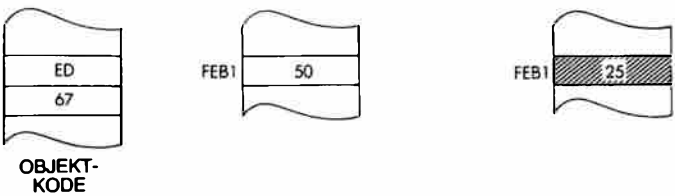
Flags:



Beispiel: RRD

Vorher: **Nachher:**

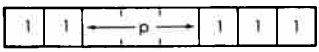




RST p Restart bei p.

Funktion: $(SP-1) \leftarrow PC_{oben}$; $(SP-2) \leftarrow PC_{unten}$; $SP \leftarrow SP-2$; $PC_{oben} \leftarrow 0$; $PC_{unten} \leftarrow p$

Format:

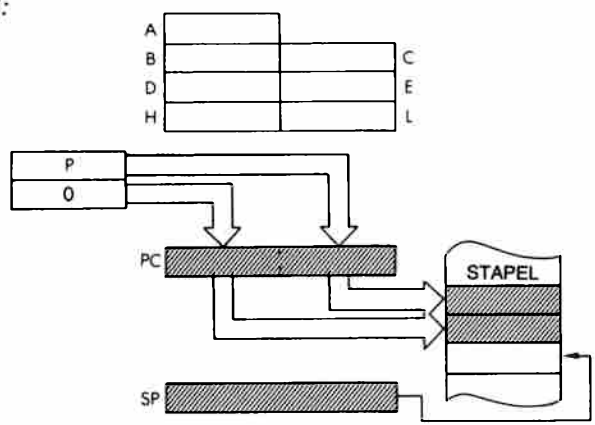


Beschreibung: Der Inhalt des Befehlszählers wird auf dem Stapel abgelegt, wie bei dem Befehl PUSH beschrieben. Dann wird der entsprechende Wert für p in den Befehlszähler geladen und der nächste Befehl von dieser neuen Adresse geholt. p kann sein:

00H – 000	20H – 100
08H – 001	28H – 101
10H – 010	30H – 110
18H – 011	38H – 111

Dieser Befehl bewirkt einen Sprung zu einer von acht Startadressen im unteren Bereich des Speichers, und er belegt nur ein einziges. Byte. Er kann als schnelle Antwort auf einen Interrupt verwendet werden.

Datenfluß:



Befehlsablauf: 3 M Zyklen; 11 T Zustände; 5,5 µsek @ 2 MHz

Adressierungsart: Indirekt.

Byte Kode:

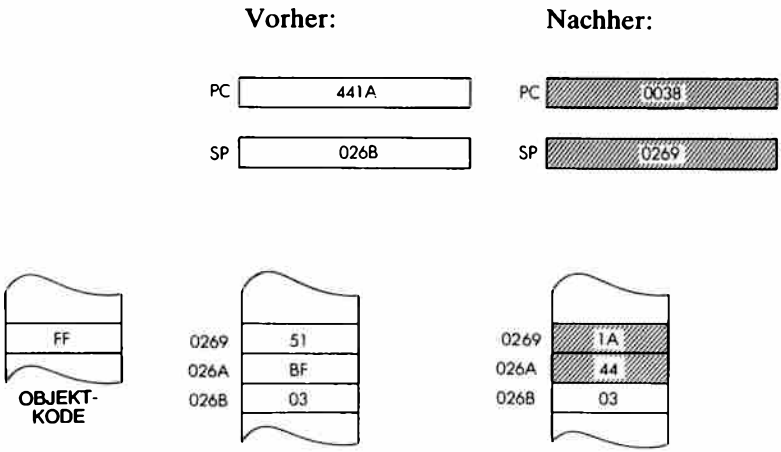
p:	00	08	10	18	20	28	30	38
	C7	CF	D7	DF	E7	EF	F7	FF

Flags:

S	Z		H		P/V	N	C

(kein Einfluß)

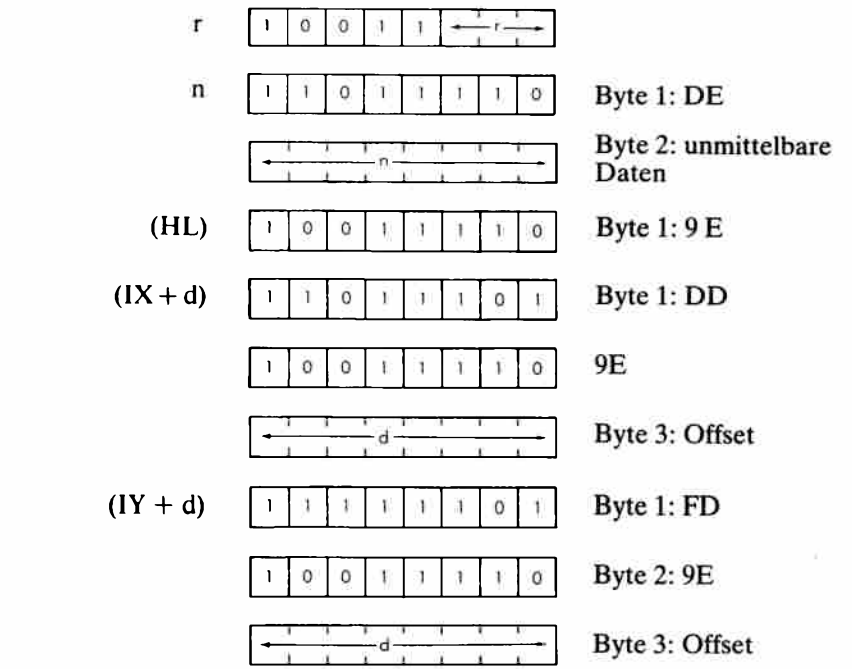
Beispiel: RST 38H



SBC A, s Subtrahiere vom Akkumulator den gegebenen Operanden s mit Übertrag.

Funktion: A ← A − s − C

Format: s: kann sein r,n,(HL),(IX + d), or (IY + d)

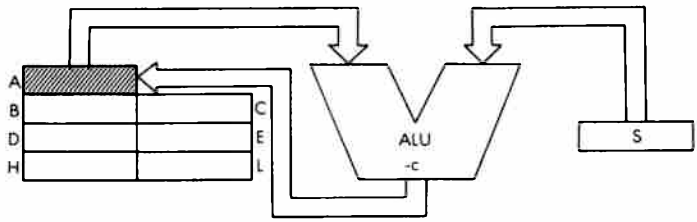


r kann sein:

A − 111	E − 011
B − 000	H − 100
C − 001	L − 101
D − 010	

Beschreibung: Der angegebene Operand s und das Übertragsflag werden vom Akkumulator subtrahiert. Das Ergebnis wird wieder im Akkumulator abgelegt. s ist bei der Beschreibung des Befehls ADD definiert.

Datenfluß:



Befehlsablauf:

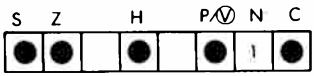
s:	*M Zyklen*	*T Zustände*	*µsek @ 2 MHz*
r	1	4	2
n	2	7	3.5
(HL)	2	7	3.5
(IX + d)	5	19	9.5
(IY + d)	5	19	9.5

Adressierungsart: r: implizit; n: unmittelbar; (HL): indirekt; (IX d),(IY d): indiziert.

Byte Kode: SBC A, r

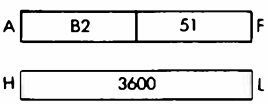
r: A	B	C	D	E	H	L
9F	98	99	9A	9B	9C	9D

Flags:

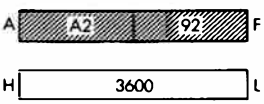


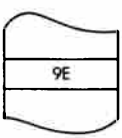
Beispiel: SBC A, (HL)

Vorher:

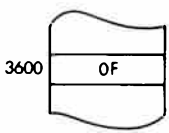


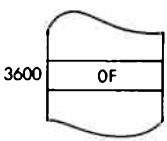
Nachher:





OBJEKT-KODE





SBC HL, ss Subtrahiere von HL das angegebene Registerpaar mit Übertrag.

Funktion: HL ← HL − ss − C

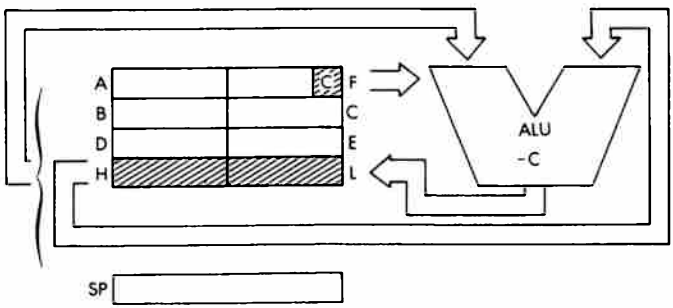
Format:

1	1	1	0	1	1	0	1	Byte 1: ED
0	1	S	S	0	0	1	0	Byte 2

Beschreibung: Der Inhalt des angegebenen Registerpaares und der Übertrag werden vom Inhalt des Registerpaars HL subtrahiert. Das Ergebnis wird wieder im Registerpaar HL abgelegt. ss kann sein:

BC − 00	HL − 10
DE − 01	SP − 11

Datenfluß:



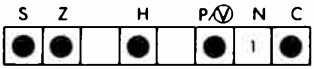
Befehlsablauf: 4 M Zyklen; 15 T Zustände; 7,5 μsek @ 2 MHz

Adressierungsart: Implizit.

Byte Kode:

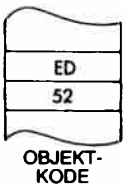
SS:	BC	DE	HL	SP
ED−	42	52	62	72

Flags:

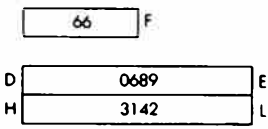


H ist gesetzt bei Übertrag von Bit 12.
C ist gesetzt bei Übertrag.

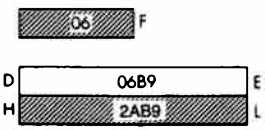
Beispiel: SBC HL, DE



Vorher:



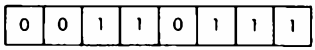
Nachher:



SCF Setze Übertragsflag.

Funktion: C ← 1

Format:



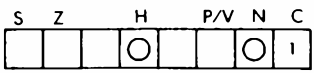
37

Beschreibung: Das Übertragsflag wird gesetzt.

Befehlsablauf: 1 M Zyklus; 4 T Zustände; 2 μsek @ 2 MHz

Adressierungsart: Implizit.

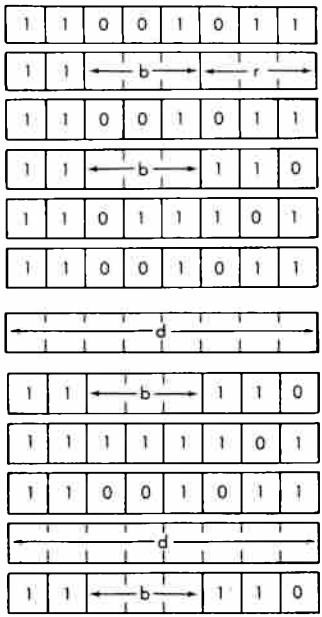
Flags:



SET b, s Setze Bit b des Operanden s.

Funktion: $s_b \leftarrow 1$

Format: *s:*



r kann sein:

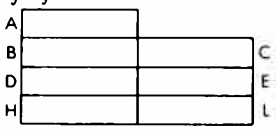
A – 111	E – 001
B – 000	H – 100
C – 001	L – 101
D – 010	

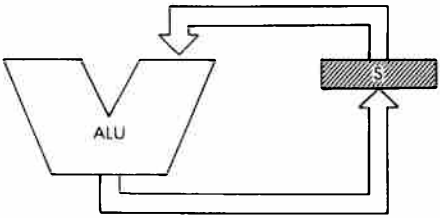
b kann sein:

0 – 000	4 – 100
1 – 001	5 – 101
2 – 010	6 – 110
3 – 011	7 – 111

Beschreibung: Das angegebene Bit von s wird gesetzt. s ist bei der Beschreibung des Befehls BIT definiert.

Datenfluß:





Befehlsablauf:

s:	*M Zyklen*	*T Zustände*	*µsek @ 2 MHz*
r	2	8	4
(HL)	4	15	7.5
(IX + d)	6	23	11.5
(IY + d)	6	23	11.5

Adressierungsart: r: implizit; (HL): indirekt; (IX d),(IY d): indiziert.

Byte Kode: SET b, r

CB-	b: r: A	B	C	D	E	H	L
	0	C7	C0	C1	C2	C3	C4	C5
	1	CF	C3	C9	CA	CB	CC	CD
	2	D7	D0	D1	D2	D3	D4	D5
	3	DF	D8	D9	DA	DB	DC	DD
	4	E7	E0	E1	E2	E3	E4	E5
	5	EF	E8	E9	EA	EB	EC	ED
	6	F7	F0	F1	F2	F3	F4	F5
	7	FF	F8	F9	FA	FB	FC	FD

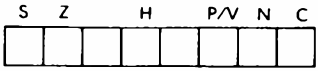
SET b, (HL)

SET b, (IX + d)

SET b, (IY + d)

b:	0	1	2	3	4	5	6	7
	C6	CE	D6	DE	E6	EE	F6	FE

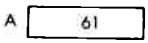
Flags:



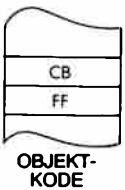
(kein Einfluß)

Beispiel: SET 7, A

Vorher:



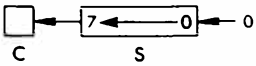
Nachher:



SLA s

Arithmetisches Links-Schieben des Operanden s.

Funktion:



Format: *s:*

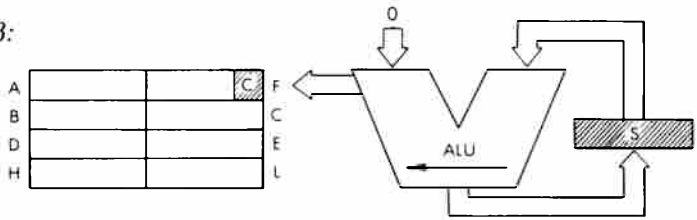
r	1 1 0 0 1 0 1 1	Byte 1: CB
	0 0 1 0 0 ← r →	Byte 2
(HL)	1 1 0 0 1 0 1 1	Byte 1: CB
	0 0 1 0 0 1 1 0	Byte 2: 26
(IX + d)	1 1 0 1 1 1 0 1	Byte 1: DD
	1 1 0 0 1 0 1 1	Byte 2: CB
	← d →	Byte 3: Offset
	0 0 1 0 0 1 1 0	Byte 4: 26
(IY + d)	1 1 1 1 1 1 0 1	Byte 1: FD
	1 1 0 0 1 0 1 1	Byte 2: CB
	← d →	Byte 3: Offset
	0 0 1 0 0 1 1 0	Byte 4: 26

r kann sein:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Beschreibung: Der Inhalt der Stelle, die durch den angegebenen Operanden festgelegt wird, wird arithmetisch links verschoben,wobei der Inhalt von Bit 7 ins Übertragsbit kommt und eine Null in Bit 0. Das Endergebnis wird wieder an der ursprünglichen Stelle abgelegt. s ist bei der Beschreibung des Befehls RLC definiert.

Datenfluß:

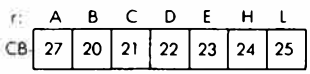


Befehlsablauf:

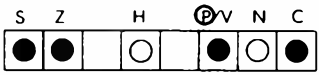
s:	*M Zyklen*	*T Zustände*	µsek @ 2 MHz
r	2	8	4
(HL)	4	15	7.5
(IX + d)	6	23	11.5
(IY + d)	6	23	11.5

Adressierungsart: r: implizit; (HL): indirekt; (IX d),(IY d): indiziert.

Byte Kode: SLA r



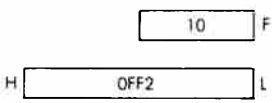
Flags:



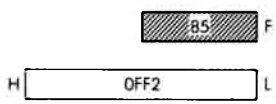
C wird von Bit 7 der Quelle gesetzt.

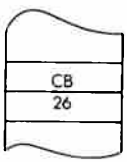
Beispiel: SLA (HL)

Vorher:

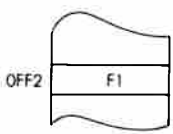


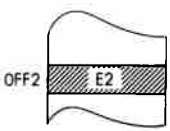
Nachher:





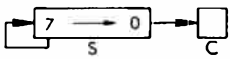
OBJEKT-KODE





SRA s Arithmetisches Rechts-Schieben des Operanden s.

Funktion:



Format:

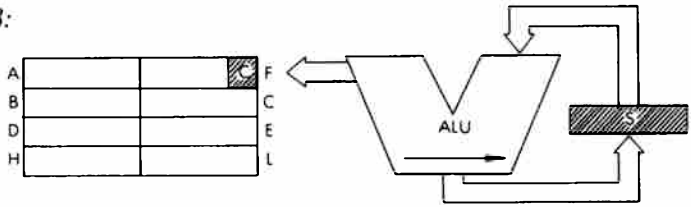
s:		
r	1 1 0 0 1 0 1 1	Byte 1: CB
	0 0 1 0 1 r r r	Byte 2
(HL)	1 1 0 0 1 0 1 1	Byte 1: CB
	0 0 1 0 1 1 1 0	Byte 2: 2E
(IX + d)	1 1 0 1 1 1 0 1	Byte 1: DD
	1 1 0 0 1 0 1 1	Byte 2: CB
	d d d d d d d d	Byte 3: Offset
	0 0 1 0 1 1 1 0	Byte 4: 2E
(IY + d)	1 1 1 1 1 1 0 1	Byte 1: FD
	1 1 0 0 1 0 1 1	Byte 2: CB
	d d d d d d d d	Byte 3: Offset
	0 0 1 0 1 1 1 0	Byte 4: 2E

r kann sein:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Beschreibung: Der Inhalt der Stelle, die durch den angegebenen Operanden festgelegt wird, wird arithmetisch rechts verschoben. Dabei kommt der Inhalt von Bit 0 ins Übertragsbit und der Inhalt von Bit 7 bleibt unverändert. Das Endergebnis wird wieder an der ursprünglichen Stelle abgelegt. s ist bei der Beschreibung des Befehls RLC definiert.

Datenfluß:



Befehlsablauf:

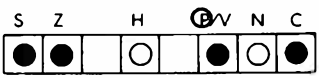
s:	*M Zyklen*	*T Zustände*	*µsek @ 2 MHz*
r	2	8	4
(HL)	4	15	7.5
(IX + d)	6	23	11.5
(IY + d)	6	23	11.5

Adressierungsart: r: implizit; (HL): indirekt; (IX d),(IY d): indiziert.

Byte Kode: SRA r

	r: A	B	C	D	E	H	L
CB-	2F	28	29	2A	2B	2C	2D

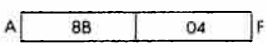
Flags:



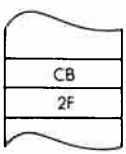
C wird von Bit 0 der Quelle gesetzt.

Beispiel: SRA A

Vorher:

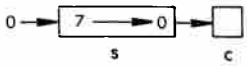


Nachher:



SRL s Logisches Rechts-Schieben des Operanden s.

Funktion:



Format:

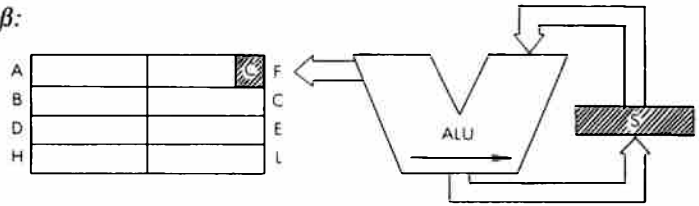
s:		Byte
r	1 1 0 0 1 0 1 1	Byte 1: CB
	0 0 1 1 1 r r r	Byte 2
(HL)	1 1 0 0 1 0 1 1	Byte 1: CB
	0 0 1 1 1 1 1 0	Byte 2: 3E
(IX + d)	1 1 0 1 1 1 0 1	Byte 1: DD
	1 1 0 0 1 0 1 1	Byte 2: CB
	d d d d d d d d	Byte 3: Offset
	0 0 1 1 1 1 1 0	Byte 4: 3E
(IY + d)	1 1 1 1 1 1 0 1	Byte 1: FD
	1 1 0 0 1 0 1 1	Byte 2: CB
	d d d d d d d d	Byte 3: Offset
	0 0 1 1 1 1 1 0	Byte 4: 3E

r kann sein:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Beschreibung: Der Inhalt der Stelle, die durch den angegebenen Operanden festgelegt wird, wird logisch rechts verschoben. Dabei kommt der Inhalt von Bit 0 ins Übertragsbit und eine Null ins Bit 7. Das Endergebnis wird wieder an der ursprünglichen Stelle abgelegt.

Datenfluß:



Befehlsablauf:

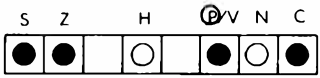
s:	*M Zyklen*	*T Zustände*	µsek @ 2 MHz
r	2	8	4
(HL)	4	15	7.5
(IX + d)	6	23	11.5
(IY + d)	6	23	11.5

Adressierungsart: r: implizit; (HL): indirekt; (IX d),(IY d): indiziert.

Byte Kode: SRL r

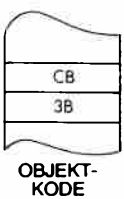
r:	A	B	C	D	E	H	L
CB	3F	38	39	3A	3B	3C	3D

Flags:

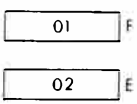


C wird von Bit 7 der Quelle gesetzt.

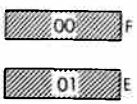
Beispiel: SRL E



Vorher:



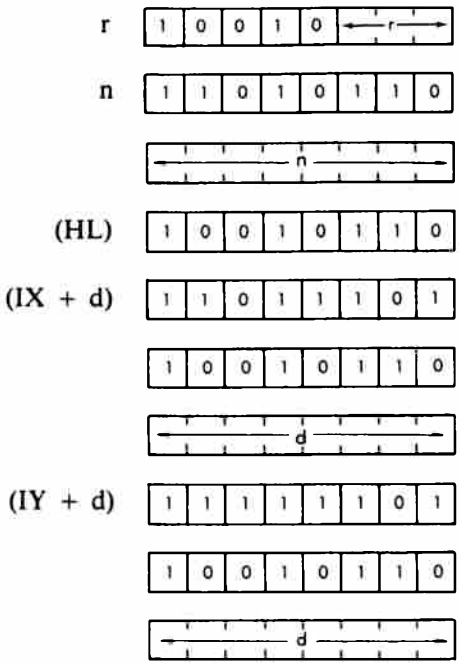
Nachher:



SUB s Subtrahiere den angegebenen Operanden s vom Akkumulator.

Funktion: A ← A − s

Format: s: kann sein r, n,(HL),(IX + d) or (IY + d)

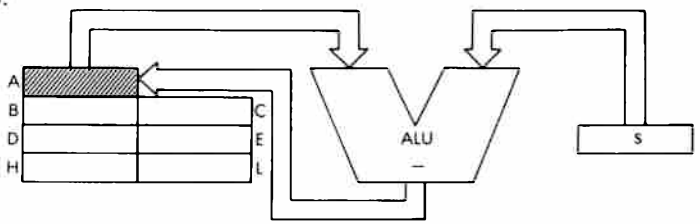


r kann sein:

A − 111	E − 011
B − 000	H − 100
C − 001	L − 101
D − 010	

Beschreibung: Der angegebene Operand s wird vom Akkumulator subtrahiert. Das Ergebnis wird wieder im Akkumulator abgelegt. s ist bei der Beschreibung des Befehls ADD definiert.

Datenfluß:

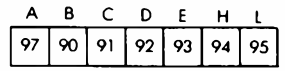


Befehlsablauf:

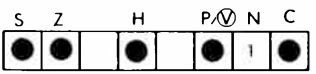
s:	*M Zyklen*	*T Zustände*	*µsek @ 2 MHz*
r	1	4	2
n	2	7	3.5
(HL)	2	7	3.5
(IX + d)	5	19	9.5
(IY + d)	5	19	9.5

Adressierungsart: **r: implizit; n: unmittelbar; (HL): indirekt; (IXX d),(IY + d): indiziert.**

Byte Kode: SUB r

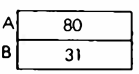


Flags:

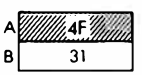


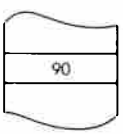
Beispiel: SUB B

Vorher:



Nachher:

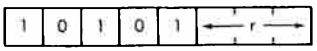


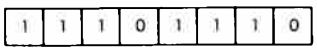


XOR s EXKLUSIVES ODER von Akkumulator und Operand s.

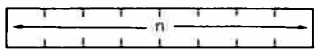
Funktion: A ← A ∀ s

Format: s: kann sein r, n,(HL),(IX + d) or (IY + d)

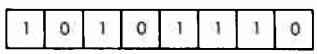
r 

n 

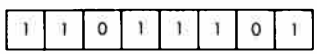
Byte 1: EE



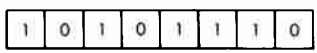
Byte 2: unmittelbare Daten

(HL) 

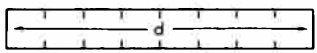
AE

(IX + d) 

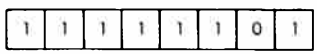
Byte 1: DD



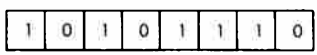
Byte 2: AE



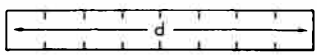
Byte 3: Offset

(IY + d) 

Byte 1: FD



Byte 2: AE



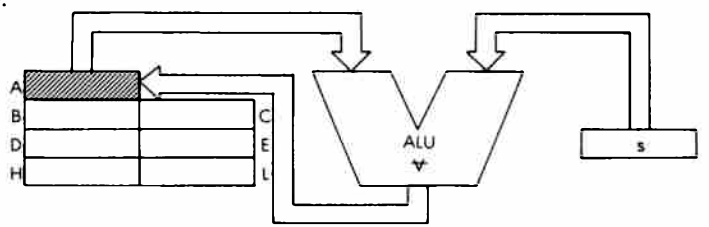
Byte 3: Offset

r kann sein:

A – 111	E – 011
B – 000	H – 100
C – 001	L – 101
D – 010	

Beschreibung: Der Akkumulator und der angegebene Operand s werden durch die logische Funktion EXKLUSIVES ODER verknüpft. Das Ergebnis wird wieder im Akkumulator abgelegt. s ist bei der Definition des Befehls ADD definiert.

Datenfluß:



Befehlsablauf:

s:	*M Zyklen*	*T Zustände*	*µsek @ 2 MHz*
r	1	4	2
n	2	7	3.5
(HL)	2	7	3.5
(IX + d)	5	19	9.5
(IY + d)	5	19	9.5

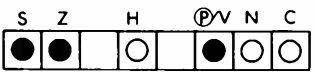
Adressierungsart: r: implizit; n: unmittelbar; (HL): indirekt; (IX d),(IY d): indiziert.

Byte Kode: XOR r

r:

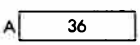
A	B	C	D	E	H	L
AF	A8	A9	AA	AB	AC	AD

Flags:

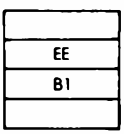


Beispiel: XOR B1H

Vorher:



Nachher:



5

Adressierungstechniken

Einführung

In diesem Kapitel wird die allgemeine Theorie der Adressierung vorgestellt und die verschiedenen Techniken, die entwickelt wurden, um das Auffinden der Daten zu erleichtern. In einem zweiten Abschnitt werden die speziellen Adressierungsarten vorgestellt, die beim Z80 verfügbar sind, zusammen mit den Vorteilen und Einschränkungen. Um den Leser schließlich mit den verschiedenen Verfahren vertraut zu machen, wird ein Abschnitt die Anwendungen möglicher Adressierungstechniken in speziellen Programmbeispielen demonstrieren.

Da der Z80 außer dem Befehlszähler noch mehrere 16-Bit-Register besitzt, mit deren Hilfe man eine Adresse festlegen kann, ist es wichtig, daß der Benutzer die verschiedenen Adressierungsarten versteht, speziell auch die Verwendung der Indexregister. Die komplizierteren Verfahren kann man am Anfang übergehen. Alle diese Adressierungsmethoden sind jedoch nützlich, wenn man Programme für diesen Mikroprozessor entwickelt. Wir wollen jetzt die verschiedenen Möglichkeiten studieren.

Die möglichen Arten der Adressierung

Adressierung ist die Art und Weise, wie der Ort des Operanden, mit dem ein Befehl arbeitet, innerhalb dieses Befehls festgelegt wird. Jetzt werden wir die hauptsächlichen Adressierungsarten untersuchen. Sie sind in Abb. 5.1 veranschaulicht.

Implizite Adressierung (oder Registeradressierung)

Befehle, die ausschließlich mit Registern arbeiten, verwenden normalerweise *implizite Adressierung.* Dies ist in Abbildung 5.1 dargestellt. Der Name impliziter Befehle leitet sich von der Tatsache ab, daß er die Adresse des Operanden, mit dem er arbeitet, nicht besonders enthält. Stattdessen legt sein Opcode ein oder mehrere Register fest, mit denen er arbeitet, üblicherweise den Akkumulator oder irgendwelche anderen

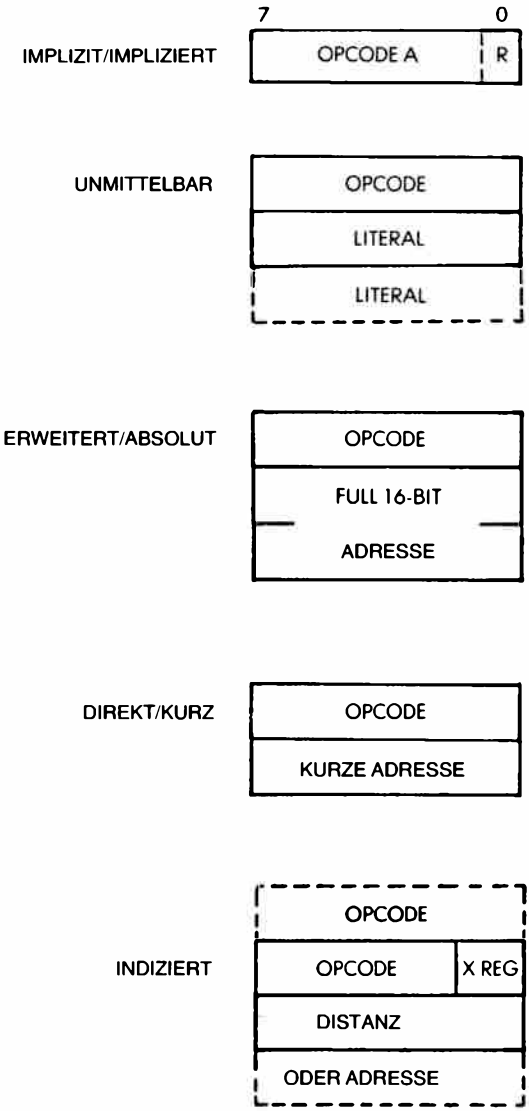


Abb. 5.1: Grundlegende Adressierungsarten

Register. Da es normalerweise nur wenige interne Register gibt (üblicherweise acht), werden dafür nur wenige Bit benötigt. Beispielsweise legen drei Bit innerhalb eines Befehls eines von acht Registern fest. Solche Befehle können deshalb normalerweise mit 8 Bit kodiert werden. Dies ist ein wesentlicher Vorteil, da ein Acht-Bit-Befehl normalerweise schneller ausgeführt wird als jeder Zwei- oder Dreibytebefehl.

Ein Beispiel für einen impliziten Befehl ist:

```
LD A,B
```

der festlegt „übertrage den Inhalt von B nach A" (Lade A aus B).

Unmittelbare Adressierung

Unmittelbare Adressierung ist in Abb. 5.1 dargestellt. Auf den 8-Bit-Opcode folgt ein 8- oder 16-Bit-Literal (eine Konstante). Diese Art von Befehl braucht man zum Beispiel, wenn man einen 8-Bit-Wert in ein Acht-Bit-Register laden will. Da der Mikroprozessor 16-Bit-Register enthält, kann es auch nötig sein, 16-Bit-Literals zu laden. Ein Beispiel für einen unmittelbaren Befehl ist:

```
ADD A,0H
```

Das zweite Wort dieses Befehls enthält das Literal „0", das zum Akkumulator addiert wird.

Absolute Adressierung

Absolute Adressierung ist das Verfahren, Daten aus dem Speicher zu holen oder in den Speicher abzulegen, bei dem die 16-Bit-Adresse der Speicherstelle direkt auf den Opcode folgt. Absolute Adressierung setzt deshalb Dreibytebefehle voraus. Ein Beispiel für absolute Adressierung:

```
LD (1234H),A
```

Dieser Befehl legt fest, daß der Inhalt des Akkumulators bei der Adresse 1234H gespeichert wird.

Der Nachteil der absoluten Adressierung ist, daß sie Dreibytebefehle nötig macht. Um die Effizienz des Mikroprozessors zu verbessern, kann eine andere Adressierungsart vorhanden sein, die nur ein Wort für die Adresse belegt: die Kurzadressierung.

Kurzadressierung

Bei dieser Adressierungsart folgt auf den Opcode eine Acht-Bit-Adresse. Auch dies ist in Abb. 5.1 dargestellt. Der Vorteil dieser Methode ist, daß sie nur zwei Byte belegt, anstatt drei Byte bei der absoluten Adressierung. Der Nachteil ist, daß alle Adressen auf einen Bereich von 0 bis 255 oder aber von −128 bis +127 beschränkt sind. Wenn man den Bereich von 0 bis 255 benutzt (die „Seite 0"), nennt man die Kurzadressierung auch Seite-Null-Adressierung. Wenn eine Kurzadressierung verfügbar ist, nennt man die absolute Adressierung oft im Gegensatz dazu

erweiterte Adressierung. Der Bereich von −128 bis +127 wird oft bei Sprungbefehlen verwendet. Dies nennt man dann relative Adressierung.

Relative Adressierung

Normale Sprung- oder Verzweigungsbefehle benötigen acht Bit für den Opcode und zusätzlich die 16-Bit-Adresse, zu der das Programm springen soll. Genau wie im vorhergehenden Beispiel hat dies den Nachteil, daß drei Worte, d. h. drei Speicherzyklen benötigt werden. Um effizientere Sprünge zu erhalten, verwendet die *relative Adressierung* nur ein Zweibyteformat. Das erste Wort legt den Sprung fest, üblicherweise zusammen mit dem Test, den er enthält. Das zweite Wort ist die Distanz. Weil die Distanz positiv oder negativ sein kann, kann man mit einem relativen Sprung um 127 Plätze vorwärts (sieben Bit) oder um 128 Plätze zurückspringen (normalerweise +129 bis −126, da der Befehlszähler schon um 2 inkrementiert wurde). Da die meisten Schleifen kurz sind, kann man oft die relative Adressierung verwenden und erhält eine deutlich verbesserte Leistungsfähigkeit für solche kurzen Routinen. Wir haben z. B. schon den Befehl JR NC,dd verwendet, der einen „Sprung, falls kein Übertrag" festlegt, innerhalb eines Bereichs von 127 Stellen (genauer von +129 bis −126).

Die zwei Vorteile der relativen Adressierung sind verbesserte Leistungsfähigkeit (es werden weniger Bytes verwendet, deswegen höhere Geschwindigkeit) und Verschieblichkeit des Programms (Unabhängigkeit von absoluten Adressen).

Indizierte Adressierung

Die indizierte Adressierung ist eine Technik, die man verwendet, wenn man nacheinander auf Elemente eines Blocks oder einer Tabelle zugreifen will. Dies wird später in diesem Kapitel an Beispielen verdeutlicht. Das Prinzip der indizierten Adressierung ist es, daß der Befehl sowohl ein Indexregister als auch eine Adresse festlegt. Der Registerinhalt und die Adresse werden addiert und ergeben die endgültige Adresse. So könnte die Adresse der Anfang einer Tabelle im Speicher sein. Das Indexregister würde dann dazu benutzt, auf alle Elemente einer Tabelle nacheinander auf effiziente Art zuzugreifen. (Dazu muß es Inkrementier-/Dekrementierbefehle für das Indexregister geben.) Praktisch bestehen oft Einschränkungen, die die Größe des Indexregisters oder die des Adreß- oder Distanzfeldes beschränken können.

Direkte und indirekte Indizierung

Man kann zwei Arten der Indizierung unterscheiden. Die direkte Indizierung ist die normale Art der Indizierung, bei der sich die endgültige Adresse ergibt als Summe von Distanz bzw. Adresse und dem Inhalt des Indexregisters. In Abb. 5.2 ist dies gezeigt, wobei ein 8 Bit langes Distanzfeld und ein 16-Bit-Indexregister angenommen sind.

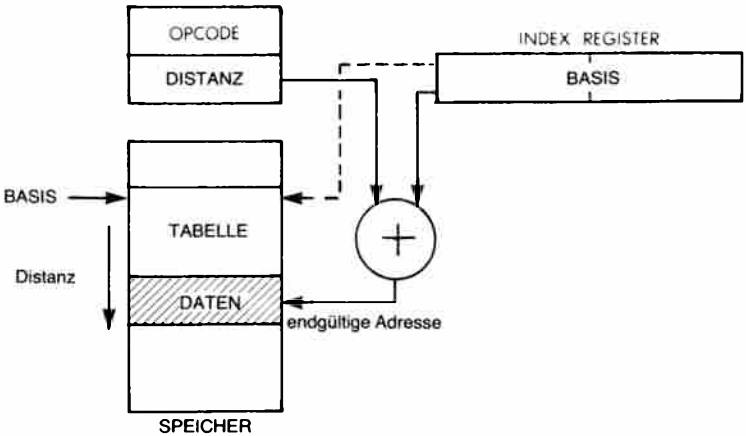


Abb. 5.2: Adressierung (direkt indiziert)

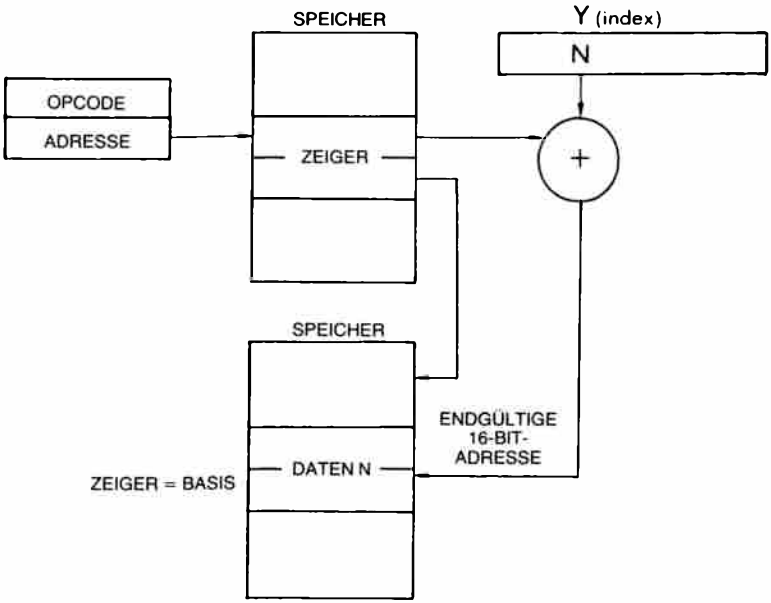


Abb. 5.3: Indirekte indizierte Adressierung

Indirekte Indizierung behandelt den Inhalt des Distanzfeldes als Adresse der wirklichen Distanz, nicht selbst schon als Distanz. Dies ist in Abb. 5.3 dargestellt. Bei der indirekten Indizierung ergibt sich die endgültige Adresse als Summe aus dem Inhalt des Indexregisters und dem Inhalt der Speicherzelle, auf die das Distanzfeld zeigt. Diese Möglichkeit bietet in der Tat eine Kombination von indirekter und indizierter Adressierung. Wir haben aber die indirekte Adressierung noch gar nicht definiert. Das wollen wir gleich nachholen.

Indirekte Adressierung

Wir haben schon gesehen, daß zwei Unterprogramme eventuell große Datenmengen austauschen wollen, die im Speicher stehen. Allgemeiner mögen zwei Programme oder zwei Unterprogramme auf einen gemeinsamen Block von Information zugreifen müssen. Wenn man das Programm allgemein halten will, kann man einen solchen Block nicht an einer festen Adresse speichern. Zum Beispiel mag sich die Größe des Blocks dynamisch vergrößern oder verkleinern, und der Block könnte abhängig von der Größe in verschiedenen Speicherbereichen stehen müssen. Es wäre deshalb nicht durchführbar, wenn man auf diesen Block mit absoluter Adressierung zugreifen wollte, ohne das Programm jedes Mal zu ändern.

Die Lösung dieses Problems besteht darin, daß man die Anfangsadresse des Blocks bei einer festen Speicheradresse ablegt. Dies ist analog zu der Situation, wenn verschiedene Leute in ein Haus hineinkommen müssen, aber nur ein Schlüssel existiert. Es wird vereinbart, den Schlüssel unter der Fußmatte zu verstecken. Jeder weiß dann, wo er nachschauen muß (unter der Fußmatte), um den Hausschlüssel zu finden (oder vielleicht

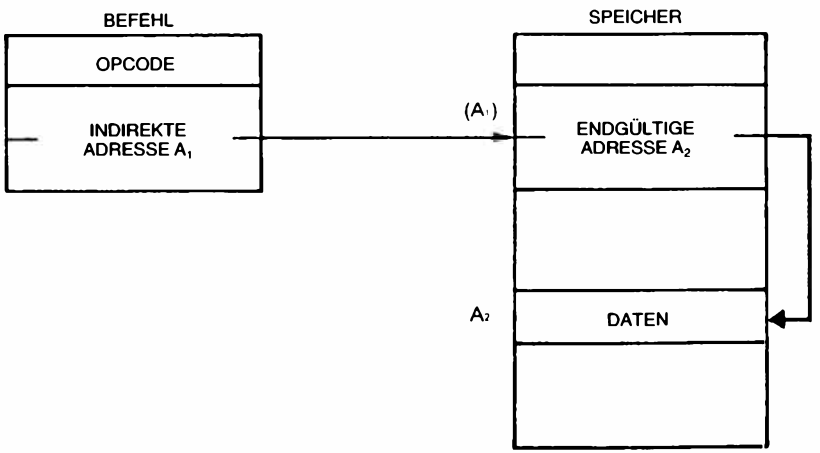


Abb. 5.4: Indirekte Adressierung

die Adresse des vorgesehenen Treffpunkts, um eine strengere Analogie zu bilden). Die indirekte Adressierung benutzt deshalb einen Opcode (16 Bit beim Z80), auf den eine 16-Bit-Adresse folgt. Diese Adresse dient dazu, ein Wort aus dem Speicher zu erhalten. Üblicherweise wird dies ein 16-Bit-Wort im Speicher sein (zwei Byte in unserem Fall), da es eine Adresse ist. Dies ist in Abb. 5.4 dargestellt. Die beiden Bytes an der festgelegten Adresse A1 enthalten „A2". A2 wird dann als die tatsächliche Adresse der Daten interpretiert, auf die man zugreifen will.

Indirekte Adressierung ist unter anderem immer dann nützlich, wenn man Zeiger verwendet. Verschiedene Teile des Programms können dann diese Zeiger verwenden, um auf ein Wort oder einen Datenblock bequem und elegant zuzugreifen. Die endgültige Adresse kann man auch erhalten, wenn innerhalb des Befehls auf ein Register verwiesen wird, in dem diese Adresse steht. Dies nennt man dann „Register indirekt".

Kombination der Adressierungsarten

Die obigen Adressierungsarten können auch kombiniert werden. Zum Beispiel sollte es in einem vollständigen allgemeinen Adressierungsschema möglich sein, mehrere indirekte Ebenen zu verwenden. Die Adresse A2 könnte wieder als indirekte Adresse verwendet werden, usw.

Indizierte Adressierung kann auch mit indirekter Adressierung kombiniert sein. Dies erlaubt einen effizienten Zugriff auf das Wort in einem Block von Daten, vorausgesetzt man weiß, wo der Zeiger auf die Anfangsadresse steht (siehe Abb. 5.2).

Jetzt haben wir uns mit allen Adressierungsarten vertraut gemacht, die üblicherweise in einem System zur Verfügung stehen können. Die meisten Mikroprozessorsysteme verfügen nicht über alle möglichen Adressierungsverfahren, sondern nur über einen kleinen Teil davon, da die Komplexität der MPU, die in einem einzigen Baustein untergebracht sein muß, begrenzt ist. Der Z80 verfügt über einen guten Teil der Möglichkeiten. Wir wollen sie jetzt untersuchen.

Die Adressierungsarten des Z80

Implizierte Adressierung (Z80)

Die implizierte Adressierung wird im wesentlichen von den Einbytebefehlen benutzt, die mit internen Registern arbeiten. Wenn implizierte Befehle nur mit internen Registern arbeiten, benötigen sie nur einen Zyklus zur Ausführung.

Beispiele für Befehle, die implizite (oder „Register-") Adressierung verwenden, sind: LD r,r'; ADD A,r; ADC A,s; SUB s; SBC A,s; AND s; OR s; XOR s; CP s; INC r.

Zilog unterscheidet noch zwischen „Registeradressierung" und „implizierter Adressierung". Implizierte Adressierung meint dann nach dieser Definition nur Befehle, die kein bestimmtes Feld enthalten, das auf ein internes Register zeigt. Damit wird eine weitere Adressierungsart eingeführt. Dies ist ein Grund, warum die Anzahl der Adressierungsarten kein geeignetes Kriterium ist, wenn man die Fähigkeiten eines Mikroprozessors charakterisieren will.

Unmittelbare Adressierung (Z80)

Da der Z80 sowohl Register einfacher Länge (acht Bit) als auch doppelter Länge (16 Bit) hat, verfügt er über zwei Arten von unmittelbarer Adressierung, mit 8-Bit- und mit 16-Bit-Literals. Die Befehle sind dann entweder ein Byte oder zwei Byte lang. Das zweite (und manchmal das dritte) Byte enthalten die Konstante oder das Literal, die in ein Register geladen oder in einer Operation verwendet werden soll. Ausnahmen sind LD IX und LD IY, die 16-Bit-Opcodes haben.

Beispiele für Befehle, die unmittelbar adressieren, sind:

LD r,n (zwei Byte),
LD dd,nn (drei Byte) und
ADD A,n (zwei Byte).

Wenn das Literal zwei Byte lang ist, nennt man die Adressierung beim Z80 „unmittelbar erweitert".

Absolute oder „erweiterte" Adressierung (Z80)

Laut Definition belegt die absolute Adressierung drei Byte. Das erste Byte ist der Opcode und die nächsten beiden Bytes sind die Adresse, die die Speicherzelle festlegen (die „absolute" Adresse).

Im Gegensatz zur „Kurzadressierung" (Achtbitadresse), nennt man dieses Verfahren auch „erweiterte Adressierung".

Beispiele für Befehle mit erweiterter Adressierung sind:

LD HL,(nn) und JP nn

wobei nn die 16-Bit-Adresse darstellt und (nn) den Inhalt dieser Adresse.

Modifizierte Seite-Null-Adressierung

Eine Seite-Null-Adressierung ist auf dem Z80 nicht vorhanden, außer für den RST-Befehl. Die spezielle Adressierungsart, die dieser Befehl verwendet, wird „modifizierte Seite-Null-Adressierung" genannt.

Der RST-Befehl enthält an den Stellen b b b ein 3-Bit-Feld, das auf eine von drei Adressen in der Seite Null zeigt. Die effektive Adresse, die in PC geladen wird, ist b b b 000. Da dieser Befehl nur ein einziges Byte belegt, wird er schnell ausgeführt und kann leicht hardwaremäßig erzeugt werden. Er wurde allgemein verwendet, um verschiedene Interrupts (maximal 8) zu bearbeiten. Sein Nachteil ist, daß entweder das be-

arbeitende Programm auf eine Länge von 16 Bit begrenzt ist, oder daß ein Sprung nötig ist, der den Geschwindigkeitsvorteil zunichte macht. Dies liegt daran, daß die 8 Adressen nur jeweils 16 Bit auseinander liegen.

Dieser Befehl wird jetzt weniger oft verwendet, seit Prioritäts-Interrupt-Controller Bausteine (PICs) auf dem Markt sind (siehe C201 und C207 für eine genauere Beschreibung von PICs). Ein PIC gibt als Antwort auf einen Interrupt Acknowledge automatisch einen drei Byte langen absoluten Sprung aus.

Dieser Befehl wird jetzt allgemein als Restart verwendet.

Relative Adressierung (Z80)

Relative Adressierung belegt definitionsgemäß zwei Byte. Das erste ist der Opcode, und das zweite gibt die Distanz mit Vorzeichen an.

Um diese Adressierungsart von einem absoluten Sprung zu unterscheiden, wird sie „JR" abgekürzt.

Bezüglich der Ausführungszeit ist dieser Befehl mit Vorsicht zu behandeln. Immer wenn ein Test fehlschlägt, d. h. wenn keine Verzweigung eintritt, dauert dieser Befehl nur sieben Taktzyklen lang. Der Grund dafür ist, daß der Befehlszähler schon auf den Befehl zeigt, der als nächster ausgeführt werden soll.

Trifft jedoch der Test zu, d. h. wenn der Sprung ausgeführt wird, benötigt der Befehl 12 „T-Zyklen". Eine neue effektive Adresse muß berechnet und in den Befehlszähler geladen werden. Wenn man nicht weiß, ob der Sprung ausgeführt wird oder nicht, muß man die Tatsache berücksichtigen, daß der Sprung entweder 12 T-Zyklen (Bedingung erfüllt) oder 7 T-Zyklen dauern kann (Bedingung nicht erfüllt).

Wenn man eine Schleife mit einem relativen Sprung (JR) entwirft, dann wird diese schneller ausgeführt, wenn man eine Bedingung testet, die normalerweise nicht erfüllt ist, z. B. die Bedingung nicht-Null für den Zähler.

Verwendet man JRs außerhalb von Schleifen und ist die getestete Bedingung unbekannt, nimmt man für die Dauer von JR oft einen Durchschnittswert an.

Dieses Problem der Ausführungszeit tritt bei dem unbedingten Sprung JR e nicht auf. Dabei wird keine Bedingung getestet, und der Befehl dauert immer 12 T-Zyklen.

Indizierte Adressierung (Z80)

Diese Adressierungsart existierte beim 8080 nicht, und sie wurde beim Z80 hinzugefügt (wie auch die beiden Indexregister). Dabei wurde es nötig, dem Opcode ein weiteres Byte hinzuzufügen, so daß daraus beim Z80 ein 16-Bit-Opcode wurde (LDIR ist ein anderes Beispiel für einen 16-Bit-Opcode). Die Struktur der indizierten Befehle ist in Abb. 5.5 gezeigt.

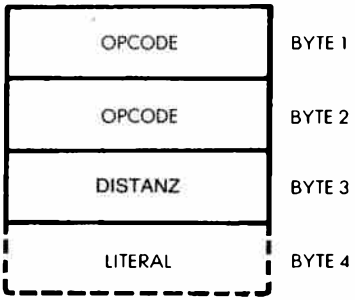


Abb. 5.5: Die indizierte Adressierung hat einen Zweibyte-Opcode

Folgende Befehle erlauben eine indizierte Adressierung:

LD, ADD, DEC, RES, INC, RLC, BIT, SET.

Dieses Adressierungsverfahren wird in Programmen, die mit Blöcken von Daten, Tabellen oder Listen arbeiten, ausgiebig verwendet werden.

Indirekte Adressierung (Z80)

Der Z80 besitzt eine beschränkte Möglichkeit, indirekt zu adressieren, genannt „Register-indirekte-Adressierung". Dabei kann jedes der 16-Bit-Registerpaare BC, DE und HL als Speicheradresse verwendet werden.

Immer wenn sie auf 16-Bit-Daten zeigen, zeigen sie auf den unteren Teil. Der obere Teil liegt in der nächsten (höheren) Adresse.

Kombinationen der Adressierungsarten

Im wesentlichen gibt es keine Kombinationen von Adressierungsarten, mit der Ausnahme, daß Befehle, die zwei Operanden ansprechen, für beide Operanden verschiedene Adressierungsarten verwenden können.

So kann ein Ladebefehl oder ein arithmetischer Befehl auf den einen Operanden unmittelbar zugreifen und auf den anderen über eine indizierte Adresse.

Auch kann der Mechanismus der Bitadressierung auf das acht Bit lange Byte in drei verschiedenen Adressierungsarten zugreifen, wie im nächsten Abschnitt beschrieben.

Die spezifischen Adressierungsverfahren, die bei jedem Befehl verfügbar sind, sind in den Tabellen im vorhergehenden Kapitel angegeben.

Bitadressierung

Die Bitadressierung bezeichnet man allgemein nicht als Adressierungsverfahren, wenn die Adressierung als Zugriff auf ein *Byte* definiert ist. Ob sie jedoch als Adressierungsart oder als Gruppe von Befehlen defi-

niert ist, es ist in jedem Fall eine nützliche Möglichkeit. Da sie in der Bezeichnungsweise von Zilog als Adressierungsverfahren definiert ist, werden wir sie hier beschreiben. Sie ist für den Z80 spezifisch und war beim 8080 nicht verfügbar.

Bitadressierung bezieht sich auf das Verfahren, auf einzelne Bits zuzugreifen. Der Z80 besitzt spezielle Befehle, um festgelegte Bits in einer Speicherzelle oder in einem Register zu setzen, zurückzusetzen und zu testen. Das festgelegte Byte kann mit einem der folgenden Verfahren adressiert werden: Register, Register-indirekt und indiziert. Innerhalb des Opcodes werden drei Bits verwendet, um eines von acht Bit auszuwählen.

Der Gebrauch der Z80-Adressierungsarten

Lange und kurze Adressierung

In verschiedenen Programmen, die wir entwickelt haben, haben wir schon relative Sprünge benutzt. Sie erklären sich selbst. Eine interessante Frage ist: Was kann man tun, wenn der mit dem relativen Sprung erreichbare Bereich für unsere Zwecke nicht ausreicht. Eine einfache Lösung ist es, einen sogenannten *langen Sprung* zu verwenden. Dies heißt einfach, auf eine Adresse zu springen, in der ein absoluter oder „langer" Sprung steht.

```
JR NC,$+3       Sprung auf aktuelle Adresse
                +3, wenn C gelöscht
JP FAR          Springe sonst zu FAR
(nächster Befehl)
```

Dieses zweizeilige Programm bewirkt einen Sprung nach FAR, wenn das Carry (Übertragsbit) gesetzt ist. Dies löst unser Problem mit dem langen Sprung. Deshalb wollen wir uns jetzt den komplizierteren Adressierungsverfahren zuwenden, d. h. der indizierten und der indirekten Adressierung.

Die Anwendung der Indizierung für den sequentiellen Zugriff auf einen Block

Indizierung wird hauptsächlich angewendet, um auf aufeinanderfolgende Adressen in einer Tabelle nacheinander zuzugreifen. Eine Einschränkung ist, daß die maximale Länge kleiner als 256 sein muß, so daß die Distanz in ein Acht-Bit-Register paßt.

Wir haben gelernt, wie man auf ein Zeichen testet. Jetzt wollen wir eine Tabelle von 100 Elementen auf das Zeichen „*" durchsuchen. Die Anfangsadresse dieser Tabelle wird BASE genannt. Die Tabelle hat nur

100 Elemente. Das Programm ist unten angegeben (siehe das Flußdiagramm in Abb. 5.6):

```
SEARCH  LD    IX,BASE
        LD    A,„*“
        LD    B,COUNT
TEST    CP    (IX)
        JR    Z,FOUND
        INC   IX
        DJNZ  TEST
NOTFND  ...
```

Ein verbessertes Programm wird unten im Abschnitt über den Blocktransfer vorgestellt.

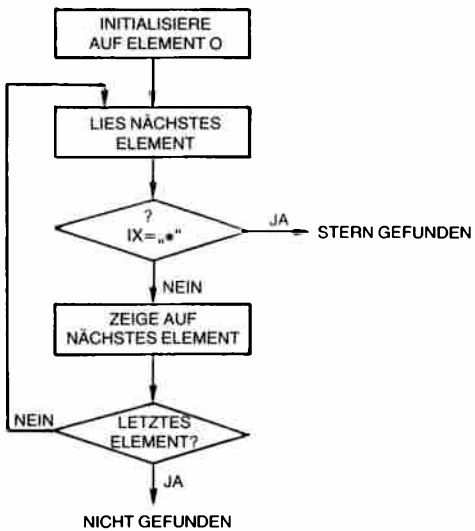


Abb. 5.6: Flußdiagramm zur Zeichensuche

Ein Blocktransfer-Programm für weniger als 256 Elemente

Wir wollen die Anzahl der Elemente in dem Block, der kopiert werden soll, COUNT nennen. Es wird angenommen, daß diese Anzahl kleiner als 256 ist. FROM ist die Anfangsadresse des Blocks. TO ist die Anfangsadresse des Speicherbereichs, in den der Block kopiert werden soll. Der Algorithmus ist ganz einfach: Wir übertragen jeweils ein Wort. Um zu wissen, welches Wort wir übertragen haben, speichern wir seine Posi-

tion im Zähler C. Das Programm ist unten angegeben:

```
BLKMOV LD   IX,FROM
       LD   IY,TO
       LD   C,COUNT
NEXT   LD   A,(IX)      Hole Wort
       LD   (IY),A
       INC  IX
       INC  IY
       DEC  C
       JR   NZ,NEXT
```

Wir wollen es überprüfen:

```
BLKMOV LD   IX,FROM
       LD   IY,TO
       LD   C,COUNT
```

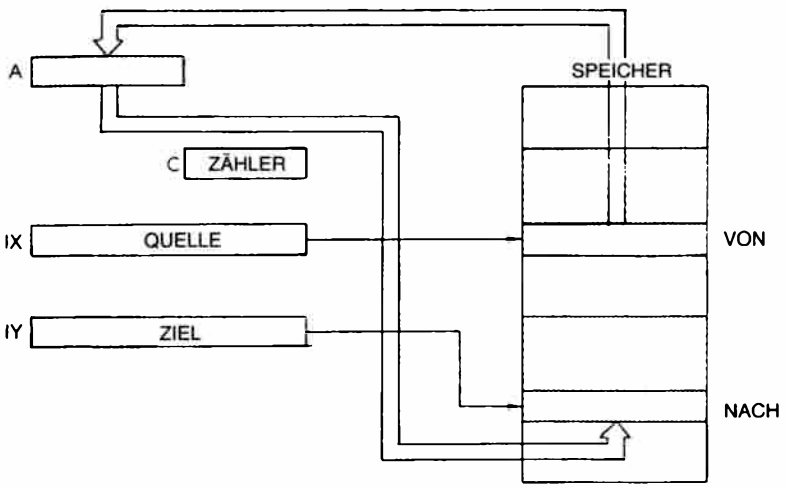


Abb. 5.7: Blocktransfer – Initialisierung der Register

Diese drei Befehle initialisieren die Register IX, IY und C, wie in Abb. 5.7 veranschaulicht. Das Indexregister IX dient als Zeiger auf die Quelle und wird regelmäßig inkrementiert. Das Register C wird mit der Anzahl der Elemente geladen, die kopiert werden sollen (begrenzt auf 256, da dies ein Acht-Bit-Register ist), und wird regelmäßig dekrementiert. Sobald C auf Null dekrementiert wurde, sind alle Elemente verschoben. Die beiden nächsten Befehle:

```
NEXT   LD   A,(IX)
       LD   (IY),A
```

laden den Inhalt der Speicherzelle, auf die IX zeigt, in den Akkumulator

und übertragen ihn dann an die Stelle des Speichers, auf die IY zeigt. Mit anderen Worten, diese beiden Befehle übertragen ein Wort aus dem Quellenblock in den Zielblock. Dann werden die beiden Indexregister inkrementiert:

```
INC  IX
INC  IY
```

und das Zählerregister wird dekrementiert:

```
DEC  C
```

Schließlich kehrt die Programmschleife zur Marke NEXT zurück, solange der Zähler nicht Null ist:

```
JR   NZ,NEXT
```

Dies ist ein Beispiel für eine mögliche Verwendung der Indexregister. Wir wollen es jedoch mit einem gleichwertigen Programm vergleichen, das für einen anderen Mikroprozessor geschrieben wurde, für den 6502 von MOS Technology. Er besitzt auch die Möglichkeit zur Indizierung, verwendet aber andere Vereinbarungen (d. h. er hat andere Einschränkungen der allgemeinen Indexmöglichkeiten). Das Programm ist unten angegeben:

```
        LDX  NUMBER
NEXT    LDA  FROM,X
        STA  TO,X
        DEX
        BNE  NEXT
```

Ohne auf Einzelheiten des obigen Programms einzugehen, wird der Leser sofort bemerken, um wieviel kürzer es ist als das vorhergehende. Dies liegt daran, daß das Indexregister X als variable Distanz verwendet wird, während FROM und TO feste Quell- und Zieladressen sind.

Dieses Beispiel sollte herausstellen, daß Indizierung nicht notwendigerweise zu effizientem Kode führt, obwohl dies in der Theorie ein leistungsfähiges Verfahren ist. Dies liegt daran, daß die verschiedenen Mikroprozessoren die Adressierung unterschiedlich einschränken. Wirkliche universelle Indizierung setzt die Möglichkeit eines 16-Bit langen Distanz- oder Adreßfelds genauso voraus wie ein 16-Bit-Indexregister.

Es sollte jedoch angemerkt werden, daß dieses spezielle Problem beim Z80 mit vier Spezialbefehlen gelöst wird. Ein universeller Blocktransfer wird jetzt beschrieben, der mit nur vier Befehlen aufgebaut werden kann. Um jedoch dem Z80 gerecht zu werden, wollen wir dem Leser zusätzliche Aufgaben vorschlagen:

Aufgabe 5.1: Schreiben Sie ein Blocktransfer-Programm für den Z80 im Stil des obigen 6502-Programms, d. h. nehmen Sie an, daß das Indexregister eine Distanz enthält. Nehmen Sie an, daß Quelle und Ziel in der Seite Null liegen, d. h. im Adreßbereich 0 bis 256. Natürlich sei vorausgesetzt, daß die Anzahl der Elemente in jedem Block so klein ist, daß sie sich nicht überlappen.

Aufgabe 5.2: Nehmen Sie jetzt an, daß Quelle- und Zielbock irgendwo im Speicher leiegen, allerdings in derselben Seite. Schreiben Sie das obige Programm für diesen Fall um.

Allgemeines Blocktransfer-Programm (mehr als 256 Elemente)

Abbildung 5.8 zeigt die Registerzuordnung und die Speicherbelegung. Das Programm ist unten angegeben:

```
LD   BC,COUNT Anzahl Bytes
LD   DE,TO    Zieladresse
LD   HL,FROM  Quelladresse
LDIR          Übertrage alle Bytes
```

Belegter Speicher: 11 Byte
Zeitbedarf: 21 Taktzyklen/Byte

Der erste Befehl ist:

```
LD   BC,COUNT
```

Er lädt die Anzahl der Elemente, die übertragen werden sollen (eine 16-Bit-Zahl), ins Registerpaar BC. Die nächsten beiden Befehle initialisieren das Registerpaar DE und entsprechend das Registerpaar HL:

```
LD   DE,TO
LD   HL,FROM
```

Der vierte Befehl schließlich:

```
LDIR
```

führt die vollständige Übertragung durch.

LDIR ist ein *automatischer Blocktransferbefehl.* Seine Leistungsfähigkeit sollte nach diesem Beispiel klar sein. LDIR führt folgenden Ablauf aus: Der Inhalt der Speicherstelle, auf die HL zeigt, wird in die Speicherzelle übertragen, auf die DE zeigt: (DE)←(HL). Danach wird DE inkre-

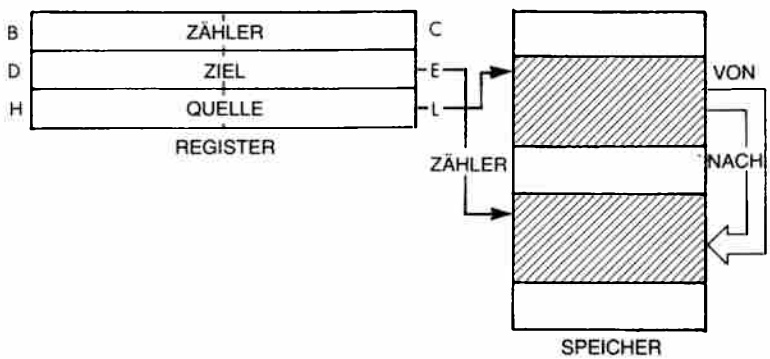


Abb. 5.8: Speicherbelegung beim Blocktransfer

mentiert: DE=DE+1 und HL inkrementiert: HL=HL+1. Danach wird BC dekrementiert: BC=BC−1. Wird BC dabei 0, dann ist der Befehl beendet. Ansonsten wird der Befehl wiederholt.

Der Nutzen und die Leistung des Befehls LDIR sollte an dieser Stelle ohne weitere Kommentare klar sein. Auf ähnliche Art kann unsere Suche nach dem Zeichen „Stern" durch Verwendung eines automatischen Befehls, des Spezialbefehls CPIR beim Z80, verbessert werden. Das entsprechende Programm erscheint unten:

```
        LDA  ,'*'
        LD   BC,COUNT
        LD   HL,STRING
        CPIR
        JR   Z,STAR
NOSTAR  ...
```

Der erste Befehl lädt den Kode des Zeichens Stern in den Akkumulator. Dann wird das Registerpaar BC als Zähler für die Zahl der Worte initialisiert, die in dem Block durchsucht werden sollen:

```
LD   BC,COUNT
```

Das Registerpaar H und L wird auf die Anfangsadresse des Blocks gesetzt, der durchsucht werden soll (STRING). Dann wird der automatische Befehl ausgeführt:

```
LD   HL,STRING
CPIR
```

Der Befehl CPIR ist ein automatischer Vergleichsbefehl. Der Inhalt der Speicherstelle, deren Adresse in H und L steht, wird mit dem Inhalt des Akkumulators verglichen. Ergibt sich Gleichheit, dann wird das Z-Flag auf 1 gesetzt. Dann wird das Registerpaar HL inkrementiert und BC dekrementiert. Der Befehl wird solange wiederholt, bis entweder dasRegisterpaar BC den Wert 0 erreicht, oder bis sich eine Gleichheit ergibt. Deshalb muß man nach Ausführung des Befehls das Z-Flag testen, um festzustellen, ob der Vergleich erfolgreich war (in Extremfällen kann man ja 64 k durchsucht haben, ohne daß das Zeichen auftrat). Dies ist die Aufgabe des letzten Befehls:

```
JR   Z, STAR
```

Aufgabe 5.3: Schreiben Sie das obig Programm so um, daß die Suche in umgekehrter Reihenfolge abläuft. (Hinweis: Verwenden Sie den Befehl CPDR.)

Wir wollen jetzt ein Programm entwickeln, das die Eigenschaften der beiden vorangehenden Programme kombiniert. Wir wollen einen Blocktransfer von der Stelle FROM zur Stelle TO ausführen, der automatisch dann gestoppt wird, wenn das Zeichen „Stern" gefunden wird.

Das Programm ist unten angegeben:

```
        LD   BC,COUNT
        LD   HL,FROM
        LD   DE,TO
        LD   A,'*'        Zeichen „Ende"
TEST    CP   (HL)         Vergleich mit Speicher
        JR   Z,END        Ende bei Gleichheit
        LDI               Übertrage Zeichen und ändere
                          Zeiger
        JR   PO,TEST      Mache weiter, bis Test beendet
                          P/V gibt an, ob BC=0
```

Die ersten drei Befehle des Programms führen die übliche Initialisierung durch. Sie setzen das Zählerregister und die Zeiger auf Quelle und Ziel:

```
        LD   BC,COUNT
        LD   HL,FROM
        LD   DE,TO
```

Der Stern wird „wie üblich" im Akkumulator abgelegt, so daß er mit dem Zeichen verglichen werden kann, das aus dem Speicher gelesen wird.

```
        LD   A,'*'
```

Genau das tut der nächste Befehl:

```
TEST    CP   (HL)
```

Ob der Vergleich Gleichheit ergab, stellt man fest, indem man das Z-Flag testet. Bei Gleichheit ist das Z-Flag gesetzt. Der nächste Befehl führt das durch:

```
        JR   Z,END
```

Der nächste Befehl führt einen *automatischen Transfer* aus:

```
        LDI
```

Dieser Befehl überträgt das Zeichen und ändert die Zeiger und den *Zähler* in einem einzigen Befehl. LDI überträgt den Inhalt der Speicherstelle, auf die H und L zeigen, in die Speicherstelle, auf die D und E zeigen: (DE)=(HL). DE und HL werden inkrementiert:

```
DE = DE + 1
HL = HL + 1
```

Schließlich wird BC dekrementiert: BC wird zu BC−1. Eine Besonderheit dieses Befehls ist es, daß das Flag P/V zurückgesetzt wird, wenn BC auf „0" dekrementiert wird, und sonst gesetzt wird. Dieses wird im letzten Befehl ausdrücklich getestet, um festzustellen, ob das Programm beendet werden soll:

```
        JR   PO,TEST
```

Addition zweier Blöcke

Hier werden wir ein Programm entwickeln, das zwei Blöcke element-

weise addiert. Die Blöcke sollen bei den Adressen BLK1 und BLK2 starten und die gleiche Anzahl COUNT von Elementen haben. Das Programm ist unten angegeben:

```
BLKADD LD   IX,BLK1
       LD   IY,BLK2
       LD   B,COUNT
LOOP   XOR  A
       LD   A,(IX+0)
       ADC  A,(IY+0)
       LD   (IX),A
       DEC  IX
       DEC  IY
       DEC  B
       JR   NZ,LOOP
```

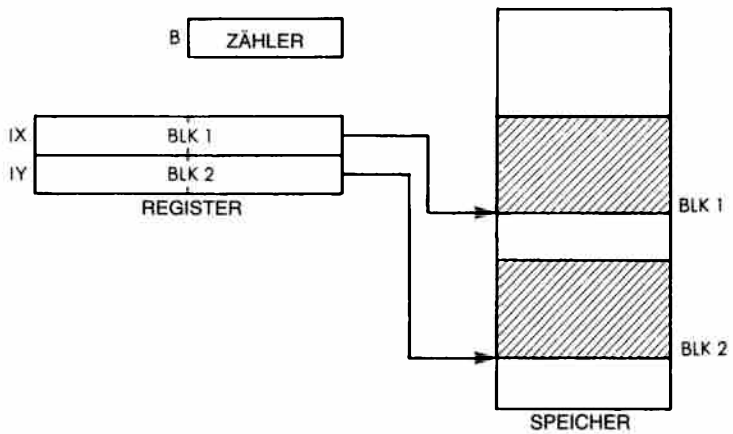


Abb. 5.9: Addition zweier Blöcke: BLK1=BLK1+BLK2

In Abb. 5.9 ist die Speicherbelegung angegeben. Das Programm ist einfach. Die Anzahl der Elemente, die addiert werden sollen, wird in das Zählerregister B geladen, und die beiden Indexregister IX und IY werden auf ihre Startwerte BLK1 und BLK2 initialisiert:

```
BLKADD LD   IX,BLK1
       LD   IY,BLK2
       LB   B,COUNT
```

Dann wird das Übertragsbit für die erste Addition gelöscht:

```
       XOR  A
```

Das erste Element wird in den Akkumulator geladen:

```
LOOP   LD   A,(IX+0)
```

Dann wird das entsprechende Element aus BLK2 dazu addiert:

```
       ADC  A,(IY+0)
```

Schließlich wird das Ergebnis in dem Element in BLK1 gespeichert:

```
LD   (IX),A
```

Die beiden Zeiger IX und IY werden dekrementiert:

```
DEC IX
DEC IY
```

und ebenso das Zählerregister B:

```
DEC B
```

Die Additionsschleife wird wiederholt ausgeführt, solange das Zählerregister nicht Null ist:

```
JR   NZ,LOOP
```

Aufgabe 5.4: Können Sie das obige Programm verwenden, um eine 32-Bit-Addition auszuführen?

Aufgabe 5.5: Können Sie das obige Programm verwenden, um eine 64-Bit-Addition auszuführen?

Aufgabe 5.6: Ändern Sie das obige Programm so, daß das Ergebnis in einem getrennten Block gespeichert wird, der bei der Adresse BLK3 beginnt.

Aufgabe 5.7: Ändern Sie das obige Programm so, daß statt der Addition eine Subtraktion ausgeführt wird.

Aufgabe 5.8: Ändern Sie das obige Programm so, daß die Adressen BLK1 und BLK2 am oberen Ende eines Blocks (bei der tieferen Adresse) und nicht am unteren Ende stehen (siehe Abb. 5.10).

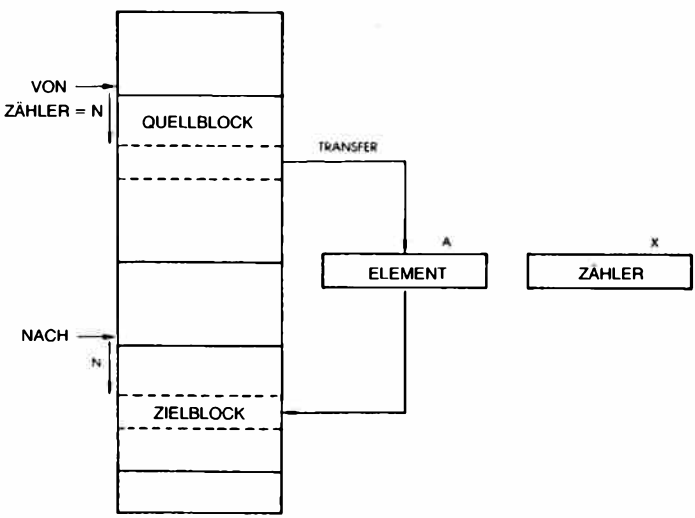


Abb. 5.10: Speicherorganisation für den Blocktransfer

Zusammenfassung

In diesem Kapitel wurde eine vollständige Beschreibung der Adressierungstechniken dargestellt. Es wurde gezeigt, daß der Z80 viele verschiedene Verfahren bietet, und die speziellen Adressierungsarten des Z80 wurden untersucht. Schließlich wurden mehrere Anwendungsprogramme vorgestellt, um den Wert der verschiedenen Adressierungsarten zu zeigen. Eine effiziente Programmierung des Z80 setzt ein Verständnis dieser Techniken voraus. Sie werden bei den Programmen, die in diesem Buch noch folgen, angewendet werden.

Aufgaben:

Aufgabe 5.9: Schreiben Sie ein Programm zur Addition der ersten 10 Byte einer Tabelle, die bei der Adresse „BASE" steht. Das Ergebnis soll 16 Bit haben. (So wird eine Prüfsumme berechnet.)

Aufgabe 5.10: Können Sie das gleiche Problem ohne Indizierung lösen?

Aufgabe 5.11: Kehren Sie die Reihenfolge der 10 Byte in dieser Tabelle um. Speichern Sie das Ergebnis bei der Adresse „REVER".

Aufgabe 5.12: Durchsuchen Sie die Tabelle nach dem größten Element. Speichern Sie es bei der Adresse „LARGE".

Aufgabe 5.13: Addieren Sie die entsprechenden Elemente der drei Tabellen mit den Anfangsadressen BASE1, BASE2 und BASE3. Die Länge dieser Adressen steht bei der Adresse „LENGTH" in Seite Null.

6

Ein-/Ausgabe-Techniken

Einführung

Wir haben bis jetzt gelernt, wie man Information zwischen dem Speicher und den verschiedenen Registern des Prozessors austauschen kann. Wir haben gelernt, die Register zu verwalten und die vielen Befehle anzuwenden, um Daten zu bearbeiten. Jetzt müssen wir lernen, mit der Außenwelt in Verbindung zu treten. Dies nennt man Eingabe/Ausgabe.

Eingabe bezieht sich auf die Aufnahme von Daten von externen Peripheriegeräten (Tastatur, Platte oder ein physikalischer Sensor). *Ausgabe* bedeutet die Übertragung von Daten vom Mikroprozessor oder vom Speicher zu externen Geräten wie Drucker, Bildschirm, Platte oder Sensor und Relais.

Wir wollen in zwei Schritten vorgehen. Zuerst wollen wir lernen, die Ein-/Ausgabeoperationen durchzuführen, die man für gebräuchliche Geräte braucht. Zweitens wollen wir lernen, verschiedene Ein-/Ausgabegeräte gleichzeitig zu *verwalten,* d. h. sie wie eine Liste zu bearbeiten. Im zweiten Teil werden u. a. die Techniken Abfrage (englisch: polling) und Unterbrechung (englisch: interrupt) vorgestellt.

Eingabe und Ausgabe

In diesem Abschnitt wollen wir lernen, einfache Signale wie Impulse zu erkennen oder zu erzeugen. Dann werden wir Techniken studieren, die einen korrekten zeitlichen Ablauf ausführen oder messen. Dann sind wir in der Lage, kompliziertere Ein-/Ausgabe-Verfahren zu verstehen, wie schnelle serielle und parallele Übertragung.

Die Ein-/Ausgabe-Befehle des Z80

Der Z80 ist mit einem speziellen Satz von Ein-/Ausgabe-Befehlen ausgestattet. Die meisten Mikroprozessoren besitzen keine speziellen Ein-/Ausgabe-Befehle, sondern verwenden die allgemeinen Befehle auch für Ein-/Ausgabegeräte. Wie der 8080 verfügt auch der Z80 über grundlegende Befehle zur Eingabe und Ausgabe. Der Z80 verfügt aber außer-

dem über zusätzliche Ein-/Ausgabe-Befehle. Diese werden hier genauer beschrieben, um das Verständnis der Programme zu erleichtern, die in diesem Kapitel im weiteren vorgestellt werden.

Die grundlegenden Befehle zur Ein-/Ausgabe sind: IN A,(n) und OUT (n),A. Der Z80 hat diese beiden Befehle vom 8080 geerbt. Sie übertragen ein Byte zwischen dem Akkumulator und dem ausgewählten Port (Ein-/Ausgabekanal), d. h. sie führen eine Lese- oder Schreiboperation aus. Die Adressierung funktioniert so, daß die Adresse „n" des Ein-/Ausgabegerätes auf der unteren Hälfte des Adreßbusses (A_0 bis A_7) erscheint, der Inhalt des Akkumulators auf der oberen (A_8 bis A_{15}). Falls ein Ein-/Ausgabegerät auch irgendwelche der Adreßleitungen A_8 bis A_{15} dekodiert und mehr als 256 Geräte adressiert werden, kann es nötig sein, den Inhalt des Akkumulators ausdrücklich auf Null zu setzen. Üblich ist dies aber nicht, d. h. in der Regel dekodieren Ein-/Ausgabegeräte nur die untere Hälfte des Adreßbusses. Auch in den folgenden Beispielen wollen wir annehmen, daß höchstens 256 Ein-/Ausgabegeräte vorhanden sind, und daß diese nicht mit den Adreßleitungen A_8 bis A_{15} verbunden sind, so daß der Inhalt des Akkumulators nichts mit der Adressierung zu tun hat und nicht z. B. vor der Verwendung des Befehls IN auf Null gesetzt werden muß.

Bei dem speziellen Eingabebefehl IN r,(C) wird der Inhalt des Registers C als Adresse des Ein-/Ausgabegerätes benutzt. Wird dieser Befehl verwendet, dann stellt das Register B automatisch die obere Hälfte des Adreßbusses (A_8 bis A_{15}). Das festgelegte Register r wird aus der Adresse in C geladen. „r" kann irgendeines der Universalregister sein.

Erzeugung eines Signals

Im einfachsten Fall wird ein Ausgabegerät vom Computer ein- oder ausgeschaltet. Um den Zustand des Ausgabegerätes zu ändern, muß der Programmierer nur eine logische Größe von „0" auf „1" oder von „1" auf „0" verändern. Wir wollen annehmen, daß ein externes Relais an Bit „0" eines „OUT1" genannten Registers angeschlossen ist. Um es einzuschalten, schreiben wir einfach eine „1" in die entsprechende Stelle des Registers. Wir nehmen hier an, daß OUT1 die Adresse dieses Ausgaberegisters in unserem System darstellt. Das folgende Programm schaltet das Relais ein:

```
TURNON LD   A,00000001B   Lade Zeichen nach A
       OUT  (OUT1),A      Ausgabe an Gerät
```

Hierbei ist OUT der Ausgabebefehl.

Wir haben angenommen, daß der Zustand der anderen sieben Bit des Registers OUT1 keine Bedeutung hat. Das ist jedoch oft nicht der Fall. An diese Bits könnten andere Relais angeschlossen sein. Deshalb wollen wir dieses einfache Programm verbessern. Wir wollen das Relais einschalten, ohne den Zustand irgendeines anderen Bits in dem Register zu verändern. Wir wollen annehmen, daß der Inhalt dieses Registers gele-

sen und eingeschrieben werden kann. Unser verbessertes Programm heißt dann:

```
TURNON IN    A,(OUT1)     Lies Inhalt von OUT1
       OR    00000001B    Setze Bit „0" in A auf „1"
       OUT   (OUT1),A
```

Das Programm liest zuerst den Inhalt der Adresse OUT1 und führt dann damit ein inklusives OR aus. Dies verändert nur das Bit an der Stelle 0 auf „1" und läßt den Rest des Registers unverändert. (Siehe Kapitel 4 für weitere Einzelheiten der Operation OR.) Dies ist in Abb. 6.1 veranschaulicht.

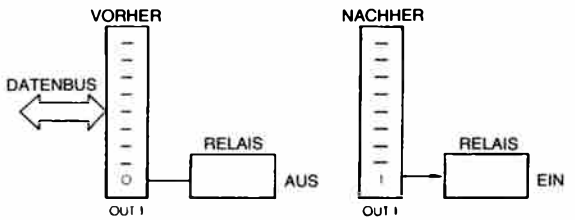


Abb. 6.1: Ein Relais einschalten

Impulse

Ein *Impuls* wird genau auf die gleiche Art erzeugt, wie der Pegel oben. Zuerst wird ein Bit eines Ausgangs eingeschaltet und dann wieder ausgeschaltet. Das Ergebnis ist ein Impuls. Dies ist in Abb. 6.2 veranschaulicht. Jetzt muß man jedoch noch ein zusätzliches Problem lösen: Man muß einen Impuls mit der richtigen zeitlichen Länge erzeugen. Dazu wollen wir überlegen, wie man eine berechnete Verzögerung erzeugt.

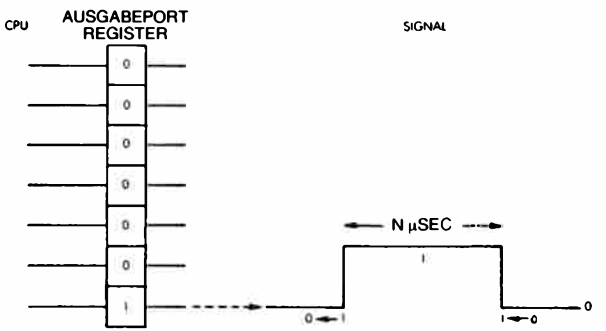


Abb. 6.2: Ein programmierter Impuls

Erzeugung und Messung einer Verzögerung

Eine Verzögerung kann man softwaremäßig oder hardwaremäßig erzeugen. Hier wollen wir untersuchen, wie man sie mit einem Programm erzeugt,und später wollen wir zeigen, wie man sie mit einem Hardwarezähler ausführt, den man programmierbaren Zeitgeber (programmable interval timer, PIT) nennt.

Eine programmierte Verzögerung erhält man durch Zählen. Ein Zählerregister wird mit einem Wert geladen und dann dekrementiert. Das Programm läuft in einer Schleife ab und dekrementiert das Zählerregister so lange, bis der Wert „0" erreicht wird. Die gewünschte Verzögerung ergibt sich durch die Zeit, die dieser Prozeß beansprucht. Als Beispiel wollen wir eine Verzögerung von 67 Taktzyklen erzeugen:

```
DELAY   LD    A,5          A ist der Zähler
NEXT    DEC   A            Dekrementieren
        JR    NZ,NEXT      Test und Sprung zu NEXT
```

Dieses Programm lädt A mit dem Wert 5. Der nächste Befehl dekrementiert A und der folgende Befehl bewirkt solange einen Sprung zu NEXT, wie A noch nicht auf „0" dekrementiert wurde. Wenn A schließlich auf Null dekrementiert wurde, wird das Programm die Schleife verlassen und den Befehl ausführen, der dann folgt. Die Logik dieses Programms ist einfach und erscheint in dem Flußdiagramm in Abb. 6.3.

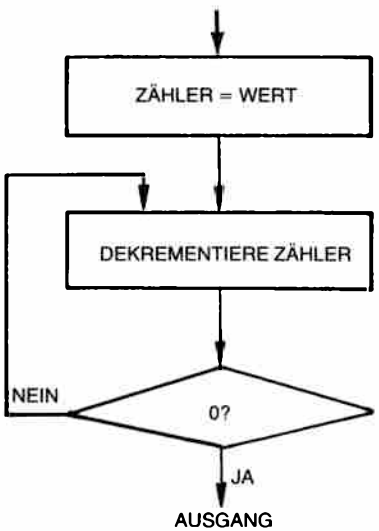


Abb. 6.3: Prinzipielles Flußdiagramm einer Verzögerungsschleife)

Wir wollen jetzt die effektive Verzögerung berechnen, die das Programm bewirkt. Dazu schauen wir in Kapitel 4 dieses Buchs nach, wie viele Taktzyklen jeder dieser Befehle braucht:
LD A,n ist unmittelbar adressiert und belegt sieben Taktzyklen. DEC braucht vier Zyklen. Schließlich braucht JR 12 Zyklen, mit Ausnahme des letzten Schrittes, bei dem er 7 Zyklen belegt. Wenn wir die Zahl der Zyklen für JR in der Tabelle nachschlagen, sehen wir, daß es zwei Möglichkeiten gibt: Tritt keine Verzweigung ein, braucht JR nur sieben Zyklen. Wird die Verzweigung aber ausgeführt, was in der Schleife meist der Fall sein dürfte, werden 12 Zyklen gebraucht.

Der Zeitbedarf ist deshalb sieben Zyklen für den ersten Befehl plus 16 Zyklen für die beiden nächsten, multipliziert mit der Anzahl der Durchläufe durch die Schleife, minus 5 Zyklen für den letzten JR, der zu keiner Verzweigung führt:

$$\text{Verzögerung} = 7 + 16 \times 5 - 5 = 82 \text{ Zyklen}$$

Wenn wir einen Zyklus von 0,5 μs annehmen, beträgt diese programmierte Verzögerung 41 Mikrosekunden.

Die Verzögerungsschleife, die beschrieben wurde, wird von den meisten Ein-/Ausgabeprogrammen verwendet. Man sollte sie gut verstehen. Versuchen Sie die folgenden Aufgaben zu lösen:

Aufgabe 6.1: Wie groß ist die maximale und die minimale Verzögerung, die mit diesen drei Befehlen eingebaut werden kann?
Aufgabe 6.2: Ändern Sie das Programm so, daß Sie eine Verzögerung von etwa 100 Mikrosekunden erhalten.

Will man eine längere Verzögerung einbauen, so ist es eine einfache Lösung, in das Programm zwischen DEC und JR zusätzliche Befehle einzubauen. Am einfachsten ist es, NOP-Befehle einzufügen. (NOP tut vier Zyklen lang nichts.)

Längere Verzögerungen

Längere Verzögerungen kann man softwaremäßig erreichen, indem man einen längeren Zähler verwendet. Man kann ein Registerpaar verwenden, um einen 16-Bit-Zähler zu speichern. Zur Vereinfachung wollen wir annehmen, daß der unterste Zählerstand „0" ist. Das unterste Byte wird mit „255" geladen, dem höchsten Zählerstand, und läuft dann durch eine Dekrementierschleife. Immer wenn auf „0" dekrementiert wurde, wird das obere Byte des Zählers um eins dekrementiert. Das Programm ist beendet, sobald das obere Byte auf „0" dekrementiert wurde. Braucht man eine größere Genauigkeit der Verzögerung, kann der unterste Zählerstand von Null verschieden sein. In diesem Fall würden wir ein Programm schreiben, wie gerade erklärt, und am Ende das dreizeilige Verzögerungsprogramm anhängen, das oben beschrieben wurde.

Ein 24-Bit Verzögerungsprogramm ist unten angegeben:

```
DEL24   LD    B,COUNTH   Oberer Zähler (8 Bit)
DEL16   LD    DE,-1
LOOPA   LD    HL,COUNTL  Unterer Zähler
LOOPB   ADD   HL,DE      Dekrementiere HL
        JR    C,LOOPB    Wiederhole, bis Null
        DJNZ  LOOPA      Dekrementiere B und springe
```

Beachten Sie, daß DE mit „−1" geladen und dazu verwendet wird, HL zu dekrementieren.

Noch längere Verzögerungen kann man natürlich dadurch erzeugen, daß man mehr als drei Worte verwendet. Dies ist analog zu der Art und Weise, wie ein Kilometerzähler im Auto funktioniert. Wenn das Rad rechts von „9" auf „0" springt, wird das nächste Rad um 1 inkrementiert. Dies ist das allgemeine Prinzip, wenn man mit mehreren getrennten Einheiten zählt.

Der hauptsächliche Nachteil dieser Methode ist es jedoch, daß der Mikroprozessor für Hunderte von Millisekunden oder gar Sekunden nichts anderes tun kann, solange Verzögerungen gezählt werden. Wenn der Computer nichts anderes zu tun hat, ist dies vollkommen akzeptabel. Im allgemeinen sollte der Mikroprozessor jedoch für andere Aufgaben zur Verfügung stehen, so daß längere Verzögerungen normalerweise nicht softwaremäßig eingebaut werden. Tatsächlich können selbst kurze Verzögerungen in einem System inakzeptabel sein, wenn in gegebenen Situationen eine bestimmte Reaktionszeit garantiert sein muß. Dann muß man eine Hardwareverzögerung verwenden. Wenn Interrupts verwendet werden, dann kann die zeitliche Genauigkeit verloren gehen, wenn die Zeitschleife unterbrochen wird.

Aufgabe 6.3: Schreiben Sie ein Programm, das eine Verzögerung von 100 ms erzeugt (für einen Fernschreiber typisch).

Hardwareverzögerung

Hardwareverzögerungen werden eingebaut, indem man einen *programmierbaren Zeitgeber* oder *„Timer"* benutzt. Ein Register des Timers wird mit einem Wert geladen. Der Unterschied ist, daß der Timer den Zähler automatisch periodisch dekrementiert. Die Periode kann normalerweise vom Programmierer bestimmt oder ausgewählt werden. Wenn der Timer auf „0" dekrementiert hat, wird er normalerweise einen Interrupt an den Mikroprozessor schicken. Er kann auch ein Zustandsbit setzen, das periodisch von dem Mikroprozessor abgefragt werden kann. Der Gebrauch von Interrupts wird später in diesem Kapitel erklärt.

Andere Betriebsarten des Timers können es sein, die Dauer eines Signals bei „0" beginnend zu zählen oder die Anzahl der empfangenen Impulse. Wenn der Timer eine Zeitspanne festlegt, so sagt man, er arbeitet in der Betriebsart *Zeitgeber*. Zählt er Impulse, so arbeitet er in der Betriebsart *Zähler*. Einige Timerbausteine können auch mehrere Register

enthalten und zusätzliche Fähigkeiten, unter denen der Programmierer wählen kann.

Die Erkennung von Impulsen

Das Problem, Impulse zu erkennen, ist die Umkehrung des Problems, Impulse zu erzeugen, und es enthält eine zusätzliche Schwierigkeit: Während ein Ausgangsimpuls unter der Kontrolle des Programms erzeugt wird, treten Eingangsimpulse *asynchron* mit dem Programm auf. Um einen Impuls zu erkennen, kann man zwei Verfahren benutzen: *Polling* und *Interrupt*. Interrupts werden später in diesem Kapitel beschrieben.

Wir wollen jetzt die Technik des Polling betrachten. Wenn man diese Technik anwendet, dann liest das Programm kontinuierlich den Inhalt eines gegebenen Registers und testet ein Bit, z. B. Bit 0. Es sei angenommen, Bit 0 sei anfänglich „0". Wird ein Impuls empfangen, dann nimmt dieses Bit den Wert „1" an. Das Programm überwacht fortlaufend das Bit 0, bis es den Wert „1" annimmt. Wenn eine „1" gefunden wird, ist der Impuls erkannt. Das Programm ist unten angegeben:

```
POLL    IN    A,(INPUT)    Lies Eingangsregister
ON      BIT   0,A          Teste auf 0
        JR    Z,POLL       Frage weiter ab, wenn 0
```

Jetzt wollen wir umgekehrt annehmen, daß die Eingangsleitung anfangs „1" ist, und daß wir eine „0" erkennen wollen. Dies ist normalerweise der Fall, wenn man den Ausgang eines Fernschreibers überwacht und das Startbit erkennen will. Das Programm ist unten angegeben:

```
POLL    IN    A,(INPUT)    Lies Eingangsregister
        BIT   0,A          Teste Bit 0
        JR    NZ,POLL      Test erfüllt?
START   ...
```

Überwachung der Pulsdauer

Die Pulsdauer kann man auf die gleiche Art und Weise überwachen, auf die man die Dauer eines Ausgangspulses berechnet. Man kann entweder eine Hardware- oder eine Softwaremethode anwenden. Wenn man einen Impuls softwaremäßig überwacht, wird ein Zähler regelmäßig um 1 inkrementiert und dann überprüft, ob der Impuls noch anliegt. Liegt der Impuls noch an, wiederholt das Programm die Schleife. Wenn der Impuls verschwindet, wird aus der Summe im Zählerregister die effektive Pulsdauer berechnet. Das Programm ist unten angegeben:

```
DURTN   LD    B,0          Lösche Zähler
AGAIN   IN    A,(INPUT)    Lies Eingang
        BIT   0,A          Überwache Bit 0
        JR    Z,AGAIN      Warte auf eine „1"
LONGER  INC   B            Inkrementiere Zähler
        IN    A,(INPUT)    Teste Bit 0
        BIT   0,A
        JR    NZ,LONGER    Warte auf eine „0"
```

Wir nehmen natürlich an, daß die maximale Pulsdauer das Register B nicht überlaufen läßt. Wäre dies der Fall, dann müßte das Programm so geändert werden, daß dies berücksichtigt würde (sonst wäre dies ein Programmierfehler!).

Da wir jetzt wissen, wie man Impulse erkennt und erzeugt, wollen wir größere Datenmengen aufnehmen oder übertragen. Zwei Fälle werden unterschieden: serielle und parallele Daten. Danach wollen wir diese Kenntnisse auf tatsächliche Ein-/Ausgabegeräte anwenden.

Parallele Übertragung von Worten

Es sei hier angenommen, daß acht Bit übertragene Daten bei der Adresse „INPUT" parallel verfügbar sind. Der Mikroprozessor muß das Datenwort bei dieser Adresse immer dann lesen, wenn ein Statuswort anzeigt, daß es gültig ist. Die Statusinformation sei in Bit 7 der Adresse „STATUS" enthalten. Wir wollen hier ein Programm schreiben, das jedes Datenwort liest und automatisch abspeichert, sobald es hereinkommt. Zur Vereinfachung wollen wir annehmen, daß die Anzahl der Worte, die gelesen werden sollen, im voraus bekannt ist und bei der Adresse „COUNT" steht. Wäre diese Information nicht verfügbar, wür-

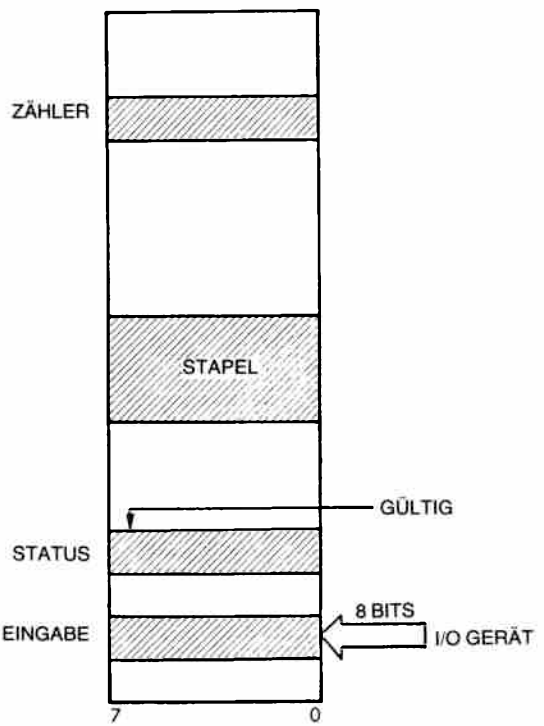


Abb. 6.4: Parallele Übertragung von Worten – der Speicher

den wir auf ein sogenanntes *Break-Zeichen* testen, wie z. B. *Rubout* oder vielleicht das Zeichen „*". Wie man dies tut, haben wir schon gelernt.

Das Flußdiagramm gibt Abb. 6.5 wieder. Es ist ziemlich einfach. Wir testen die Statusinformation, bis sie „1" wird und anzeigt, daß ein Wort bereit ist. Sobald das Wort bereit ist, lesen wir es und speichern es in einer entsprechenden Speicherstelle. Dann dekrementieren wir den Zähler und testen, ob er auf „0" dekrementiert wurde. Ist das der Fall, dann sind wir fertig; wenn nicht, dann lesen wir das nächste Wort. Ein einfaches Programm, das diesen Algorithmus ausführt, erscheint unten:

```
PARAL   LD    A,(COUNT)   Lies Zähler nach A
        LD    B,A         B ist der Zähler
WATCH   IN    A,(STATUS)  Schaue, ob Daten bereit sind
        BIT   7,A         Bit 7 ist „1", wenn Daten bereit
        JR    Z,WATCH     Daten verfügbar?
        IN    A,(INPUT)   Lies Daten
        PUSH  AF          Speichere Daten auf dem Stapel
        DEC   B           Dekrementiere Zähler
        JR    NZ,WATCH    Wiederhole so lange, bis Null
```

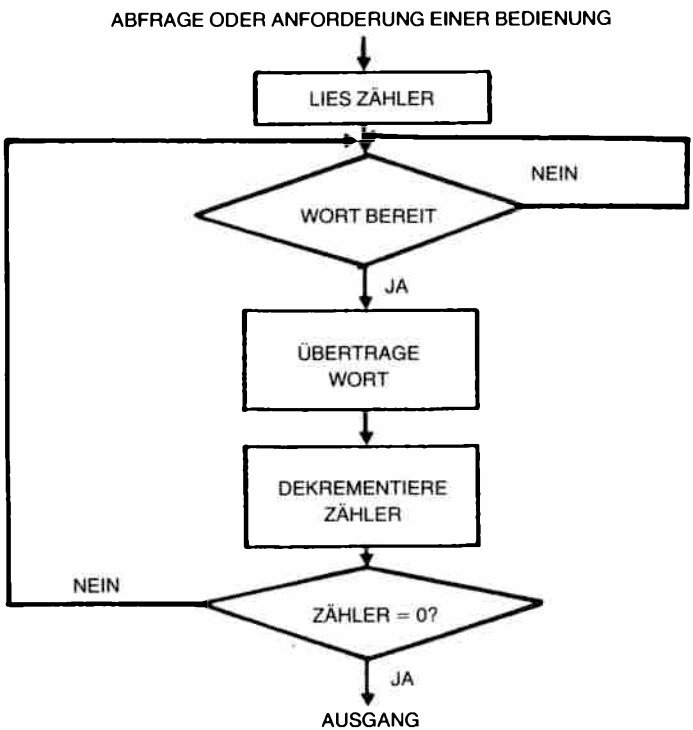


Abb. 6.5: Parallele Übertragung von Worten: Flußdiagramm

Es wurde angenommen, daß das Flag „Daten bereit" automatisch gelöscht wird, sobald STATUS gelesen wurde, wie es bei den üblichen Steuerbausteinen der Fall ist.

Die beiden ersten Befehle initialisieren das Zählerregister B:

```
PARAL   LD    A,(COUNT)
        LD    B,A
```

Beachten Sie, daß man B nicht direkt aus dem Speicher laden kann. Mann muß entweder A laden und dann seinen Inhalt nach B übertragen, oder B und C gleichzeitig laden.

Die nächsten drei Befehle des Programms lesen die Statusinformation und bewirken, daß eine Schleife so lange durchlaufen wird, wie Bit sieben des Statusregisters „0" ist. (Dies ist das Vorzeichenbit, d. h. Bit N.)

```
        IN    A,(STATUS)
        BIT   7,A          „IN" setzt die Flags nicht
        JR    Z,WATCH
```

Wird JR nicht ausgeführt, dann liegen Daten an, und wir können sie lesen:

```
        IN    A,(INPUT)
```

Das Wort wurde jetzt aus der Adresse INPUT gelesen, wo es anlag, und muß gespeichert werden. Unter der Annahme, daß genügend Platz für den Stapel vorhanden ist, können wir

```
        PUSH  AF
```

verwenden, was A (und F) auf dem Stapel ablegt. Ist der Stapel voll oder die Zahl der Worte groß, die übertragen werden sollen, dann könnten wir sie nicht auf dem Stapel ablegen und müßten sie in einen bestimmten Speicherbereich ablegen, z. B. unter Verwendung von indizierten Befehlen. Dies würde jedoch einen zusätzlichen Befehl zum Inkrementieren oder Dekrementieren des Indexregisters erforderlich machen. PUSH ist schneller (nur 11 Taktzyklen).

Das Datenwort wurde jetzt gelesen und gespeichert. Wir wollen einfach den Wortzähler dekrementieren und testen,ob wir fertig sind:

```
        DEC   B
        JR    NZ,WATCH
```

Wir erwarten so lange weitere Daten, bis der Zähler auf „0" dekrementiert wird.

Dieses Programm aus neun Befehlen kann man einen *Benchmark* nennen. Ein Benchmark-Programm ist ein sorgfältig ausgewähltes Programm, das entworfen wurde, um die Fähigkeiten eines gegebenen Prozessors für eine spezielle Anwendung zu testen. Parallele Übertragungen sind eine solche typische Anwendung. Dieses Programm wurde entworfen mit Blick auf maximale Geschwindigkeit und Effizienz. Wir wollen jetzt die maximale Übertragungsgeschwindigkeit dieses Programms berechnen. Wir wollen annehmen, daß COUNT im Speicher steht. Die

Dauer jedes Befehls steht in den Befehlsbeschreibungen in Kapitel 4. Dort finden sich folgende Werte:

```
                                  Taktzyklen
PARAL   LD      A,(COUNT)         13
        LD      B,A                4
WATCH   IN      A,(STATUS)        11
        BIT     7,A                8
        JR      Z,WATCH           7/12
        IN      A,(INPUT)         11
        PUSH    AF                11
        DEC     B                  4
        JR      NZ,WATCH          7/12
```

Die minimale Zeit für die Ausführung erhält man, wenn man annimmt, daß jedesmal Daten anliegen, wenn wir STATUS abfragen. Mit anderen Worten: Der erste Sprung wird nie ausgeführt. Dann ergibt sich folgende Zeit:

$$13 + 4 + (11 + 8 + 7 + 11 + 11 + 4 + 7) * \text{COUNT}$$

Wenn wir die ersten 17 Zyklen vernachlässigen, die zur Initialisierung des Zählerregisters nötig sind, dann benötigt die Übertragung eines Wortes 59 Zyklen oder 29,5 Mikrosekunden bei einem 2 MHz Takt.

Die maximale Datenübertragungsrate ist deshalb:

$$1 / (29{,}5 \times 10^{-6}) = 33 \text{ k Byte pro Sekunde}$$

Aufgabe 6.4: Nehmen Sie an, die Zahl der übertragenen Worte sei größer als 256. Ändern Sie das Programm entsprechend und bestimmen Sie den Einfluß auf die maximale Übertragungsrate.

Aufgabe 6.5: Verändern Sie dieses Programm, um zu versuchen, seine Geschwindigkeit zu erhöhen:

1 – durch Verwendung von JP statt JR
2 – durch Verwendung von DJNZ
3 – durch Verwendung von INIR oder INDR

War das obige Programm optimal?
Wir haben jetzt gelernt, wie man eine Parallelübertragung mit hoher Geschwindigkeit durchführt. Jetzt wollen wir einen komplizierteren Fall betrachten.

Serielle Bitweise Übertragung

Bei einem seriellen Eingang kommen die Bits der Information (Nullen oder Einsen) nacheinander auf einer Leitung herein. Diese Bits können in regelmäßigen Abständen ankommen. Das nennt man normalerweise *synchrone* Übertragung. Sie können aber auch als Gruppen von Daten in zufälligen Abständen ankommen. Dies nennt man dann *asynchrone* Übertragung. Wir wollen ein Programm entwickeln, das in beiden Fäl-

len funktioniert. Das Prinzip der Aufnahme serieller Daten ist einfach: Wir beobachten eine Eingangsleitung, die die Leitung 0 sein soll. Sobald auf dieser Leitung ein Datenbit erkannt wird, lesen wir dieses Bit ein und schieben es in ein Register. Immer wenn acht Bit zusammengekommen sind, legen wir das Datenbyte im Speicher ab und setzen dann das nächste zusammen. Um die Sache zu vereinfachen, wollen wir annehmen, daß die Anzahl der Bytes, die empfangen werden soll, im voraus bekannt ist. Sonst müßten wir beispielsweise auf ein spezielles Zeichen Break (Unterbrechung) warten und dann an dieser Stelle die bitserielle Übertragung stoppen. Wie man das tut, haben wir schon gelernt. Das Flußdiagramm für dieses Programm zeigt Abb. 6.6. Das Programm ist unten angegeben:

```
SERIAL  LD    C,0          Lösche Eingangswort
        LD    A,(COUNT)    Lade B mit der Zahl der Bytes
        LD    B,A
LOOP    IN    A,(INPUT)    Lies Port
        BIT   7,A          Bit 7 ist Status, Bit 0 Daten
        JR    Z,LOOP       Warte auf eine „1"
        SRL   A            Schiebe Datenbit ins Carry
        RL    C            Schiebe Datenbit vom Carry nach C
        JR    NC,LOOP      Solange, Bis 8 Bits vorhanden
        PUSH  BC           Speichere Wort auf dem Stapel
        LD    C,01H        Setze Markierungsbit
        DEC   B            Dekrementiere Bytezähler
        JR    NZ,LOOP      Stelle nächstes Wort zusammen
```

Dieses Programm wurde im Hinblick auf Effizienz entworfen und verwendet neue Techniken, die wir erklären wollen (siehe Abb. 6.7).

Dabei gelten folgende Vereinbarungen: Die Speicherstelle COUNT soll die Anzahl der Worte enthalten, die übertragen werden sollen. Das Register C wird dazu verwendet, acht Bits zusammenzustellen, die nacheinander hereinkommen. Die Adresse INPUT bezieht sich auf ein Eingangsregister. Es sei angenommen, daß Bit 7 dieses Registers ein Statusflag oder ein Taktbit ist. Wenn es „0" ist, liegen keine Daten an. Ist es „1", dann sind Daten verfügbar. Die Daten selbst sollen in Bit 0 der gleichen Adresse erscheinen. In vielen Fällen wird die Statusinformation in einem anderen Register abgelegt als die Daten. Es sollte eine einfache Aufgabe sein, das Programm dann entsprechend zu ändern. Zusätzlich wollen wir annehmen, daß das erste Datenbit, das von dem Programm empfangen werden soll, garantiert eine „1" ist. Es zeigt an, daß die wirklichen Daten folgen. Für den Fall, daß das nicht so wäre, werden wir später eine naheliegende Modifikation sehen, mit der das berücksichtigt wird. Das Programm entspricht genau dem Flußdiagramm in Abb. 6.6. Die ersten Zeilen des Programms bilden eine Warteschleife, die testet, ob ein Bit bereit ist. Um festzustellen, ob ein Bit bereit ist, lesen wir das Eingangsregister und testen dann das Statusbit auf Null. So lange dieses Bit „0" ist, wird der Befehl JR ausgeführt, und wir verzweigen in die

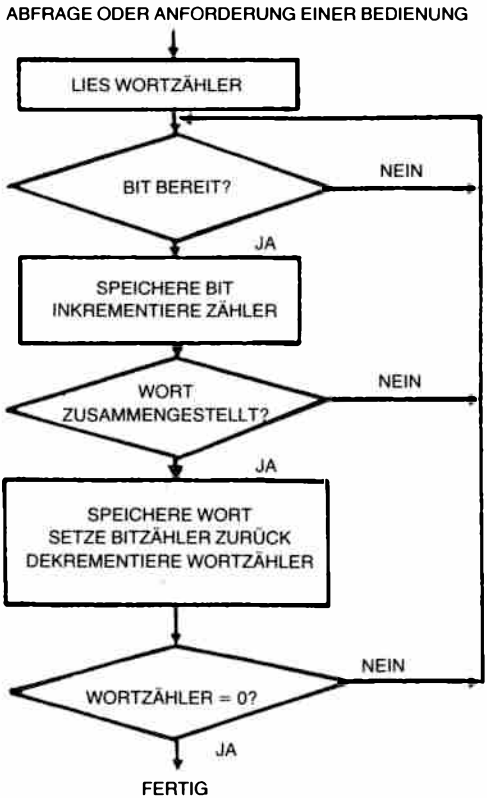


Abb. 6.6: Bitserielle Übertragung

Schleife zurück. Sobald das Statusbit (oder Taktbit) logisch wahr („1") wird, tritt der JR nicht ein und der nächste Befehl wird ausgeführt.

Diese anfängliche Folge von Befehlen entspricht dem Pfeil 1 in Abb. 6.7.

An dieser Stelle enthält der Akkumulator eine „1" in der Stelle Bit 7 und das gültige Datenbit in Stelle 0. Das erste Datenbit, das ankommen soll, wird eine „1" sein. Die folgenden Bits können jedoch „0" oder „1" sein. Wir wollen jetzt das Datenbit erhalten, das in Position 0 angekommen ist. Der Befehl:

```
SRL  A
```

schiebt den Inhalt des Akkumulators um eine Stelle nach rechts. Damit fällt das rechte Bit von A, unser Datenbit, ins Carry (Übertragsbit). Wir wollen jetzt dieses Datenbit ins Register C bringen (dieser Vorgang wird von den Pfeilen 2 und 3 in Abb. 6.7 veranschaulicht):

```
RL  C
```

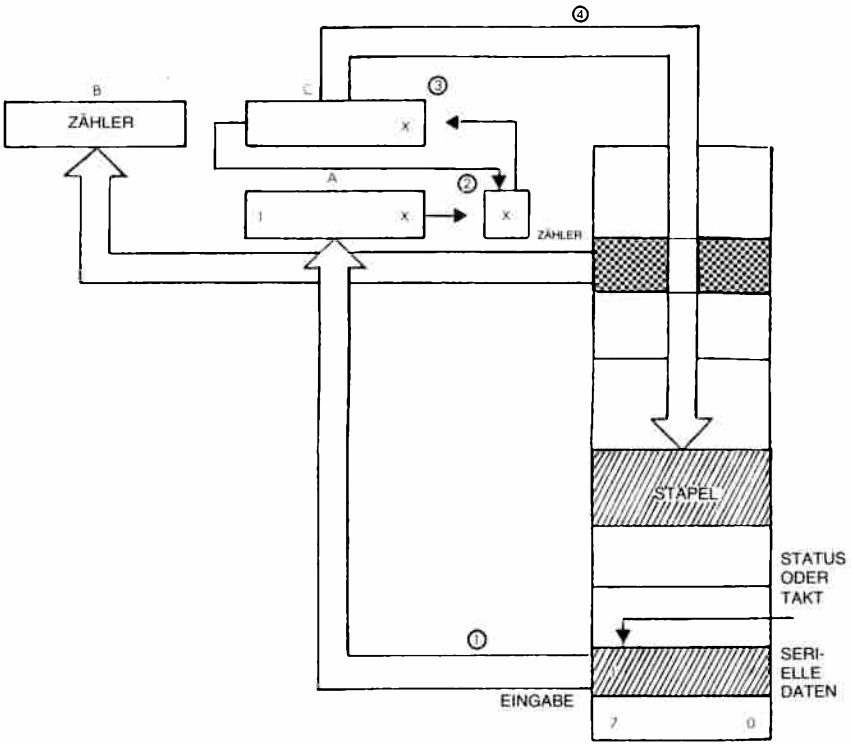


Abb. 6.7: Seriell nach Parallel: die Register

Die Wirkung dieses Befehls ist es, daß das Carrybit in die rechte Stelle von C rotiert wird. Gleichzeitig fällt das linke Bit von C ins Carrybit. (Wenn Ihnen die Rotieroperation nicht ganz klar ist, dann schauen Sie in Kapitel 4 nach!)

Es ist wichtig, sich daran zu erinnern, daß eine Rotieroperation sowohl das Carrybit rettet, hier in die rechte Stelle, und auch das Carrybit mit dem Wert von Bit 7 neu lädt.

Hier wird eine „0" ins Carry gelangen. Der nächste Befehl:

```
JR  NC,LOOP
```

testet das Carry und verzweigt zur Adresse LOOP, solange das Carry „0" ist. Dies ist unser automatischer Bitzähler. Man kann leicht sehen, daß C als Ergebnis des ersten RL C „00000001" enthält. Nach acht weiteren Schiebebefehlen wird die „1" schließlich ins Carry gelangen und die Verzweigung beenden. Dies ist eine geschickte Art, einen automatischen Schleifenzähler einzubauen, ohne daß man einen Befehl zum Dekrementieren eines Indexregisters verschwenden muß. Diese Technik

wird verwendet, um das Programm zu verkürzen und seine Effizienz zu verbessern.

Wenn JR NC schließlich nicht mehr eintritt, sind 8 Bit in C angesammelt. Dieser Wert sollte im Speicher abgelegt werden. Das wird mit dem nächsten Befehl erledigt (Pfeil 4 in Abb. 6.7):

```
PUSH  BC
```

Wir retten hier den Inhalt der Register B und C auf den Stapel. Die Speicherung auf dem Stapel ist nur möglich, wenn auf dem Stapel genug Platz ist. Unter der Voraussetzung, daß diese Bedingung erfüllt ist, ist es normalerweise die schnellste Art und Weise, ein Wort in den Speicher zu retten, obwohl ein unnötiges Register (B) gespeichert wird. Der Stapelzeiger wird automatisch auf den neuesten Stand gebracht. Würden wir das Wort nicht auf dem Stapel ablegen, dann müßten wir einen zusätzlichen Befehl verwenden, um einen Zeiger in den Speicher auf den neuesten Stand zu bringen. Wir könnten gleichwertig eine indiziert adressierte Operation ausführen, aber das würde auch voraussetzen, daß der Index dekrementiert oder inkrementiert wird, was zusätzliche Zeit kostet.

Nachdem das erste Datenwort gespeichert wurde, ist nicht länger garantiert, daß das erste hereinkommende Datenbit eine „1" ist. Es ist beliebig. Deshalb müssen wir den Inhalt auf „00000001" setzen, so daß wir es weiterhin als Bitzähler verwenden können. Dies wird mit dem nächsten Befehl durchgeführt:

```
LD  C,01H
```

Schließlich wollen wir den Wortzähler dekrementieren, da ein Wort zusammengekommen war, und testen, ob wir das Ende der Übertragung erreicht haben. Dies wird von den beiden folgenden Befehlen erledigt:

```
DEC  B
JR   NZ,LOOP
```

Das obige Programm wurde auf Geschwindigkeit ausgelegt, so daß man eine schnelle Folge von Datenbits aufnehmen kann. Sobald das Programm endet, ist es natürlich empfehlenswert, sofort die Worte aus dem Stapel zu lesen, die dort abgelegt wurden, und sie in einen anderen Speicherbereich zu übertragen. In Kapitel 2 haben wir schon gelernt, wie man einen solchen Blocktransfer ausführt.

Aufgabe 6.6: Berechnen Sie die maximale Geschwindigkeit, mit der dieses Programm serielle Datenbits lesen kann. Schauen Sie in Kapitel 4 nach, wieviele Zyklen jeder Befehl beansprucht, und berechnen Sie die Zeit, die während der Ausführung des Programms vergeht. Zur Berechnung der Zeit, die von einer Schleife verbraucht wird, multiplizieren Sie einfach die Dauer eines Durchlaufs in Mikrosekunden mit der Anzahl der Durchläufe. Nehmen Sie zur Berechnung der maximalen Geschwindigkeit an, daß immer ein Datenbit anliegt, wenn der Eingang abgefragt wird.

Dieses Programm ist schwerer verständlich als die vorhergehenden. Wir wollen es nochmals genau anschauen (siehe Abb. 6.6) und einige Besonderheiten untersuchen.

Von Zeit zu Zeit kommt ein Datenbit in Position 0 von „INPUT". Es können beispielsweise drei Einsen nacheinander ankommen. Wir müssen deshalb zwischen den Bits unterscheiden, die aufeinanderfolgend hereinkommen. Dies ist dieAufgabe des Taktsignals.

Das Signal Takt (oder STATUS) sagt uns, daß jetzt das Bit am Eingang verfügbar ist. Bevor wir ein Bit lesen, wollen wir deshalb zuerst das Statusbit testen. Ist der Status „0", müssen wir warten. Ist das Statusbit „1", dann ist das Datenbit gültig.

Wir nehmen hier an, daß das Statussignal an Bit 7 des Registers INPUT angeschlossen ist.

Aufgabe 6.7: Können Sie erklären, warum Bit 7 für den Status und Bit 0 für die Daten verwendet wird? Spielt das eine Rolle?

Sobald wir ein Datenbit eingefangen haben, wollen wir es an einem sicheren Ort aufbewahren und dann nach links schieben, so daß wir das nächste Bit aufnehmen können.

Unglücklicherweise wird der Akkumulator in diesem Programm dazu verwendet, sowohl Daten als auch den Status zu lesen und zu testen. Würden wir Daten im Akkumulator sammeln, dann würde die Stelle 7 durch das Statusbit gelöscht.

Aufgabe 6.8: Können Sie eine Methode vorschlagen, wie man den Status testen kann, ohne den Inhalt des Akkumulators zu löschen (mit einem Spezialbefehl)? Wenn man das tun kann, könnten wir dann den Akkumulator verwenden, um die Bits zu sammeln, die nacheinander hereinkommen? Können Sie die Geschwindigkeit mit einem „automatischen Sprung" erhöhen?

Aufgabe 6.9: Schreiben Sie das Programm so um, daß der Akkumulator dazu verwendet wird, die hereinkommenden Bits zu speichern. Vergleichen Sie es bezüglich Geschwindigkeit und Zahl der Befehle mit dem vorhergehenden Programm.

Wir wollen zwei weitere mögliche Varianten ansprechen.

In unserem Beispiel haben wir angenommen, daß das erste Bit, das hereinkommt, ein spezielles Signal ist, nämlich eine „1". Allgemein kann es jedoch beliebig sein.

Aufgabe 6.10: Verändern Sie das obige Programm unter der Annahme, daß das erste Bit, das hereinkommt, gültige Daten darstellt (nicht verworfen werden darf) und eine „0" oder eine „1" sein kann. Hinweis: Unser „Bitzähler" sollte weiterhin korrekt arbeiten, wenn Sie ihn mit dem richtigen Wert initialisieren.

Schließlich haben wir das zusammengestellte Wort auf den Stapel gerettet, um Zeit zu sparen. Wir hätten es natürlich auch in einem festgelegten Speicherbereich ablegen können.

Aufgabe 6.11: Modifizieren Sie das obige Programm so, daß das zusammengestellte Wort in dem Speicherbereich abgelegt wird, der bei der Adresse BASE beginnt.
Aufgabe 6.12: Verändern Sie das Programm so, daß die Übertragung endet, sobald das Zeichen „S" im Eingabefuß erkannt wird.

Die Hardware-Alternative
Wie die meisten Standardalgorithmen zur Ein-/Ausgabe kann auch dieses Verfahren hardwaremäßig realisiert werden. Den Baustein nennt man UART. Er sammelt die Bits automatisch. Will man jedoch die Zahl der Bauelemente vermindern, wird stattdessen dieses Programm oder eine Abwandlung davon verwendet.

Aufgabe 6.13: Verändern sie das Programm unter der Annahme, daß die Daten in Stelle 0 der Adresse INPUT verfügbar sind, die Statusinformation dagegen in Stelle 0 der Adresse INPUT+1.

Zusammenfassung der grundlegenden Ein-/Ausgabe
Wir haben jetzt gelernt, die grundlegenden Ein-/Ausgabe-Operationen auszuführen und auch einen Strom paralleler Daten oder serieller Bits zu verwalten. Wir sind jetzt in der Lage, mit wirklichen Ein-/Ausgabegeräten zu kommunizieren.

Kommunikation mit Ein-/Ausgabegeräten
Um Daten mit Ein-/Ausgabegeräten auszutauschen, müssen wir uns zuerst davon überzeugen, ob Daten verfügbar sind, wenn wir sie lesen wollen, oder ob das Gerät bereit ist, Daten anzunehmen, wenn wir sie ausgeben wollen. Man kann zwei Verfahren anwenden: Handshaking („Händeschütteln") und Interrupts. Wir wollen zuerst das Handshaking untersuchen.

Handshaking
Handshaking wird allgemein verwendet, um zwischen zwei beliebigen asynchronen Geräten zu kommunizieren, d. h. zwischen zwei Geräten, die nicht synchronisiert sind. Wenn wir zum Beispiel ein Wort zu einem Paralleldrucker senden wollen, müssen wir uns zuerst davon überzeugen, daß der Eingangspuffer des Druckers frei ist. Wir werden deshalb den Drucker fragen: Bist du bereit? Der Drucker antwortet dann „ja" oder „nein". Wenn er nicht bereit ist, werden wir warten. Ist er bereit, übertragen wir die Daten (siehe Abb. 6.8)

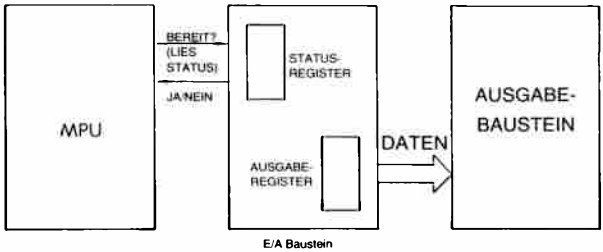


Abb. 6.8: Handshaking (Ausgabe)

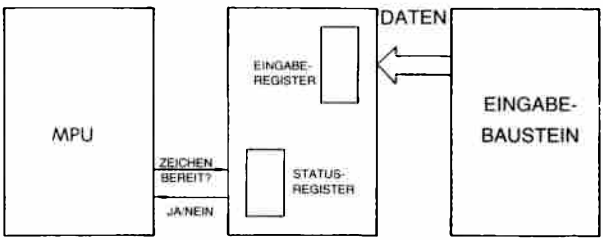


Abb. 6.8a: Handshaking (Eingabe)

Bevor wir umgekehrt Daten aus einem Eingabegerät lesen, werden wir prüfen, ob die Daten verfügbar sind. Wir werden fragen: „Sind Daten verfügbar?", und der Baustein sagt uns „ja" oder „nein". Das „ja" oder „nein" kann durch Statusbits oder mit anderen Mitteln angezeigt werden (siehe Abb. 6.8a).

Eine Analogie dazu ist es, wenn Sie mit jemandem Informationen austauschen wollen, der unabhängig ist und zur Zeit etwas anderes tun könnte. Dann sollten Sie sich davon überzeugen, daß er bereit ist, sich mit Ihnen zu unterhalten. Das übliche Höflichkeitsritual ist es, sich die Hände zu geben. Dann kann der Datenaustausch folgen. Genau dieses Verfahren wendet man bei der Kommunikation mit Ein-/Ausgabegeräten auch an.

Wir wollen dieses Verfahren jetzt an einem einfachen Beispiel veranschaulichen.

Ausgabe eines Zeichens an einen Drucker

Es sei angenommen, daß das Zeichen in der Speicherstelle CHAR steht. Das Programm zum Ausdrucken folgt unten:

```
WAIT    IN    A,(STATUS)
        BIT   7,A          Test, ob bereit
        JR    Z,WAIT       Ansonsten warten
        LD    A,(CHAR)     Hole Zeichen
        OUT   (PRNTD),A    Drucke das Zeichen
        JR    WAIT         Sprung zum nächsten Zeichen
```

Das Druckprogramm ist einfach, und es verwendet das Handshaking-Verfahren, das oben beschrieben wurde. Die Datenwege sind in Abb. 6.9 dargestellt.

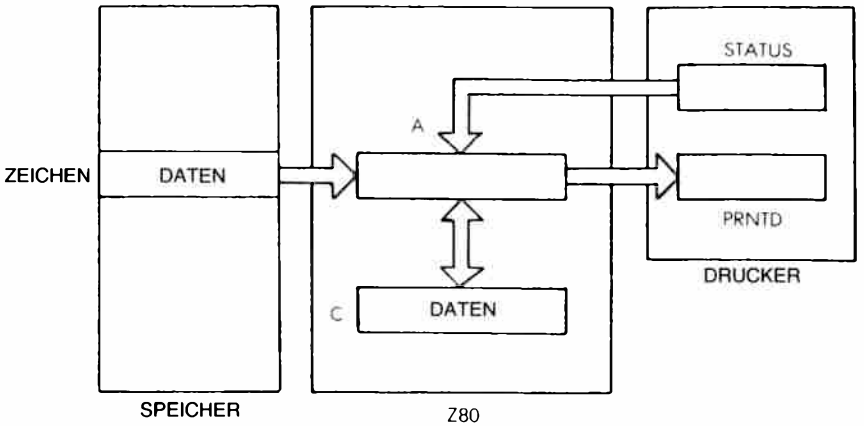


Abb. 6.9: Drucker – Datenwege

Das Zeichen (DATA genannt) steht in der Speicherzelle CHAR. Zuerst wird der Status des Druckers geprüft. Wenn Bit 7 des Statusregisters 1 wird, dann zeigt dies an, daß der Drucker zur Ausgabe bereit ist, d. h. daß sein Ausgabepuffer verfügbar ist. An dieser Stelle wird das Zeichen in den Akkumulator geladen und dann vom Akkumulator an den Drukker ausgegeben. Solange das Statusbit 0 ist, verbleibt das Programm in einer Schleife, die WAIT genannt wurde.

Aufgabe 6.14: Wieviele Befehle würden im obigen Programm eingespart, wenn es möglich wäre, Daten direkt ins Register zu laden und auch den Inhalt des Registers C direkt auszugeben?

Aufgabe 6.15: Wenn man einen wirklichen Drucker verwendet, ist es üblicherweise nötig, einen Startbefehl auszugeben, bevor man das Gerät benutzt. Verändern Sie das Programm so, daß ein solcher Befehl erzeugt wird. Nehmen Sie dazu an, daß man als Startbefehl eine 1 in die Stelle 0 des Statusregisters schreiben muß, welches bidirektional sein soll.

Aufgabe 6.16: Wenn es den Befehl BIT nicht gäbe, könnte man dann in Zeile 2 des Programms einen anderen Befehl verwenden? Erläutern Sie einen eventuellen Vorteil des Befehls BIT, falls das geht.

Aufgabe 6.17: Verändern Sie das obige Programm so, daß eine Zeichenkette aus n Zeichen ausgegeben wird, wobei n kleiner als 255 sein soll.

Aufgabe 6.18: Verändern Sie das Programm so, daß eine Zeichenkette solange ausgedruckt wird, bis das Zeichen „Carriage-Return" (Wagenrücklauf) entdeckt wird.

Wir wollen jetzt das Ausgabeverfahren komplizierter machen, indem wir eine Kodeumwandlung durchführen und gleichzeitig an mehrere Geräte ausgeben.

Ausgabe an eine Siebensegmentanzeige

Eine übliche Siebensegmentanzeige aus Leuchtdioden (LEDs) kann die Ziffern „0" bis „9" anzeigen, oder auch „0" bis „F" hexadezimal, indem mehrere der sieben Segmente leuchten. Die Abbildung 6.10 zeigt eine Siebensegmentanzeige. Die Zeichen, die man damit erzeugen kann, sind in Abb. 6.11 dargestellt.

Die Segmente der Anzeige werden in Abb. 6.10 „a" bis „g" genannt.

Eine „0" wird beispielsweise angezeigt, indem die Segmente abcdef leuchten. Wir wollen jetzt annehmen, daß Bit „0" eines Ausgabeports mit Segment „a" verbunden ist, Bit „1" mit Segment „b" usw. Bit 7 wird nicht benutzt. Um die Segmente fedcba leuchten zu lassen (zur Darstellung einer „0"), wird deshalb der Binärcode „0111111" verwendet. Hexadezimal ist dies „3F". Lösen Sie die folgende Aufgabe.

Aufgabe 6.19: Ermitteln Sie den Siebensegmentkode für die Hexadezimalziffern „0" bis „F". Füllen Sie die folgende Tabelle aus:

Hex	LED-Kode	Hex	LED-Kode	Hex	LED-Kode	Hex	LED-Kode
0	3F	4		8		C	
1		5		9		D	
2		6		A		E	
3		7		B		F	

Wir wollen jetzt Hexadezimalzahlen mit *mehreren* Siebensegmentanzeigen darstellen.

Steuerung mehrfacher LEDs

Eine LED enthält keinen Speicher. Sie zeigt die Daten nur solange an, solange Strom durch ein Segment fließt. Um die Kosten einer LED-Anzeige gering zu halten, zeigt der Mikroprozessor die Information auf jeder Stelle *nacheinander* an. Der Wechsel zwischen den Stellen muß schnell genug sein, so daß er nicht als Blinken erscheint. Die setzt voraus, daß die Zeit, die von einer LED bis zur nächsten gebraucht wird, kürzer als 100 ms ist. Wir wollen ein Programm entwerfen, das dies erledigt. Das Register C dient als Zeiger auf die Stelle, in der wir eine Ziffer anzeigen wollen. Der Akkumulator soll den hexadezimalen Wert ent-

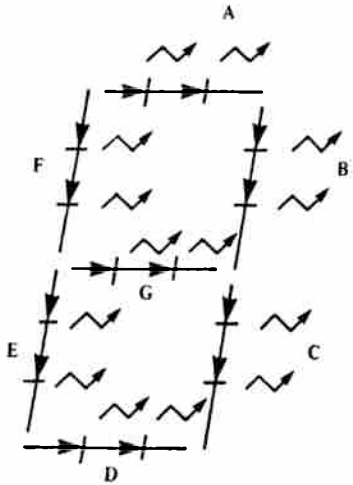


Abb. 6.10: Siebensegmentanzeige LED

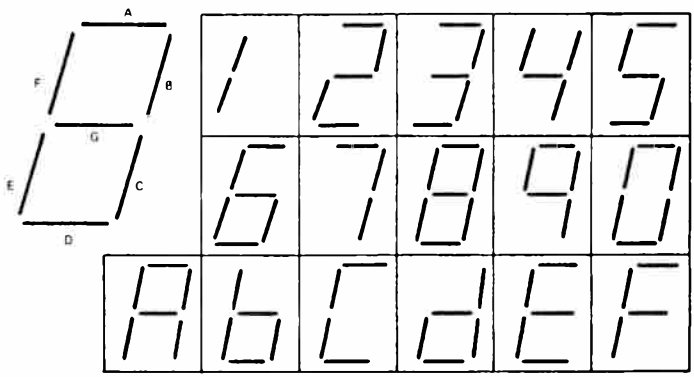


Abb. 6.11: Hexadezimalzeichen, die mit einer Siebensegmentanzeige erzeugt werden

halten, der an dieser Stelle angezeigt werden soll. Unsere erste Aufgabe ist es deshalb, den hexadezimalen Wert in seine Siebensegmentdarstellung umzuwandeln. Im vorhergehenden Abschnitt haben wir die Umwandlungstabelle aufgestellt. Da wir auf eine Tabelle zugreifen, wollen wir indizierte Adressierung verwenden, wobei der hexadezimale Wert als Distanz verwendet wird. Dies heißt, daß man den Siebensegmentkode für die Hexadezimalziffer „3" dadurch erhält, daß man in der Ta-

belle das dritte Element nach der Basis betrachtet. Die Adresse der Basis wird SEGBAS genannt. Das Programm erscheint unten:

```
LEDS    LD    E,A          A enthält die Hexadezimalziffer
        LD    D,0          „DE" wird als Distanz verwendet
        LD    HL,SEGBAS    „HL" dient als Index
        ADD   HL,DE        Adresse in der Tabelle
        LD    A,(HL)       Lies Kode aus der Tabelle
        LD    B,50H        Wert für Verzögerung
                           = beliebige große Zahl
        OUT   (C),A        Ausgabe für die festgesetzte
                           Dauer
DELAY   DEC   B            Zähler für Verzögerung
        JR    NZ,DELAY     Schleife
        DEC   C            C ist die Nummer des Ports
        LD    A,C
        CP    MINLED       Schon letzte Stelle?
        JR    NZ,OUT
        LD    BC,(MAXLED)  Falls ja, setze C auf oberste Stelle
OUT     RET
```

Das Programm setzt voraus, daß das Register C die Adresse der Stelle enthält, die als nächste leuchten soll, und daß der Akkumulator A den Wert enthält, der angezeigt werden soll.

Das Programm ermittelt zuerst den Siebensegmentkode, der dem hexadezimalen Wert im Akkumulator entspricht. Die Register D und E dienen als Distanzfeld, und die Register H und L werden als 16-Bit Indexregister verwendet. Der hexadezimale Kode wird zur Basisadresse der Tabelle addiert:

```
LEDS    LD    E,A          7-Segmentkode
        LD    D,0
        LD    HL,SEGBAS
        ADD   HL,DE
```

Danach ist eine Verzögerungsschleife eingebaut, so daß der Kode, den man aus der Tabelle erhält, für die entsprechende Dauer angezeigt wird. Hier wurde die hexadezimale Konstante „50" willkürlich gewählt:

```
        LD    A,(HL)       Lies Kode aus der Tabelle
        LD    B,50H
```

Zur Verzögerung dient eine klassische Schleife. Der erste Befehl

```
        OUT   (C),A
```

gibt den Inhalt des Akkumulators an den Ein-/Ausgabeport aus, auf den das Register C zeigt (die Nummer der Stelle). Die nächsten beiden Befehle bilden die Schleife:

```
DELAY   DEC   B
        JR    NZ,DELAY
```

Sobald die Verzögerungsschleife abgearbeitet ist, müssen wir einfach

den Zeiger auf die angezeigte Stelle dekrementieren und dafür sorgen, daß wir wieder zur höchsten Stelle verzweigen, sobald die Adresse der tiefsten Stelle erreicht ist:

```
        DEC   C
        LD    A,C
        CP    MINLED
        JR    NZ,OUT
        LD    BC,(MAXLED)
OUT     RET
```

Das obige Programm wurde als Unterprogramm geschrieben, und der letzte Befehl ist deshalb ein RET: „Rücksprung vom Unterprogramm".

Aufgabe 6.20: Es ist üblicherweise notwendig, die Segmenttreiber für die entsprechende Stelle einzuschalten, bevor die Ziffer angezeigt wird. Ändern Sie das obige Programm, indem Sie die dazu nötigen Befehle einfügen (geben Sie dazu „00" aus, bevor das Zeichen ausgegeben wird).

Aufgabe 6.21: Was würde mit der Anzeige passieren, wenn die Marke DELAY eine Zeile nach oben verschoben würde. Würde dies den zeitlichen Ablauf verändern? Würde das Aussehen der Anzeige verändert?

Aufgabe 6.22: Sie werden bemerkt haben, daß die ersten vier Befehle des Programms in Wirklichkeit eine indizierte 16-Bit-Adressierung bilden. Ohne Verwendung des Mechanismus der Indizierung wirkt dies jedoch scheinbar umständlich. Nehmen Sie an, daß die Adresse SEGBAS im voraus bekannt ist. Der obere Teil dieser Adresse werde SEGBSH genannt, der untere SEGBSL. Speichern Sie SEGBSH im oberen Teil des Registers IX. Schreiben Sie jetzt das obige Programm um und verwenden Sie die indizierte Adressierung des Z80, wobei SEGBSL als Distanzfeld des Befehls dienen soll. Was sind die Vor- und Nachteile dieses Verfahrens?

Aufgabe 6.23: Wir wollen annehmen, daß das obige Programm als Unterprogramm verwendet wird. Dann werden Sie bemerken, daß die Register B, D, E, H und L intern verwendet werden, und daß ihr Inhalt geändert wird. Wenn das Unterprogramm die Speicherzellen T1, T2, T3, T4 und T5 frei verwenden darf, können Sie dann am Anfang und am Ende des Programms Befehle einfügen, die gewährleisten, daß der Inhalt der Register B, D, E, H und L der gleiche bleibt wie bei Beginn des Unterprogramms?

Aufgabe 6.24: Es soll das gleiche Problem wie oben gelöst werden, allerdings soll der Speicherbereich T1 usw. nicht für das Unterprogramm zur Verfügung stehen. (Hinweis: Es gibt in jedem Computer einen eingebauten Mechanismus, um Information in zeitlicher Reihenfolge abzulegen.)

Wir haben bisher allgemeine Probleme der Ein-/Ausgabe gelöst. Wir wollen uns jetzt mit einem gebräuchlichen Peripheriegerät beschäftigen, mit dem Fernschreiber.

Eingabe und Ausgabe mit dem Fernschreiber

Der Fernschreiber ist ein serielles Gerät. Er sendet und empfängt Worte von Information in einem seriellen Format. Jedes Zeichen ist im 8-Bit-ASCII-Format dargestellt (die ASCII-Tabelle befindet sich am Schluß dieses Buchs). Zusätzlich geht jedem Zeichen ein „Startbit" voraus und es wird von zwei „Stopbits" abgeschlossen. In dem sogenannten 20 mA-Stromschleifen-Interface, das üblicherweise benutzt wird, ist der Zustand der Leitung normalerweise eine „1". Damit wird dem Prozessor angezeigt, daß die Leitung nicht unterbrochen ist. Ein Startbit ist ein Übergang von „1" auf „0". Es zeigt dem empfangenden Baustein an, daß Datenbits folgen. Der Standard-Fernschreiber verarbeitet 10 Zeichen in der Sekunde. (Dies gilt für amerikanische Fernschreiber, die Fernschreiber im europäischen Telexnetz benutzen einen anderen Kode, den 5-Bit-Baudot-Kode, und eine andere Übertragungsrate, nämlich 50 Bit pro Sekunde.) Wir haben bereits festgelegt, daß jedes Zeichen 11 Bit einnimmt. Dies heißt, daß ein Fernschreiber 110 Bit pro Sekunde überträgt. Man sagt, er ist ein 110-Baud-Gerät. Wir wollen ein Programm entwerfen, das serielle Bits mit der richtigen Geschwindigkeit an einen Fernschreiber ausgibt.

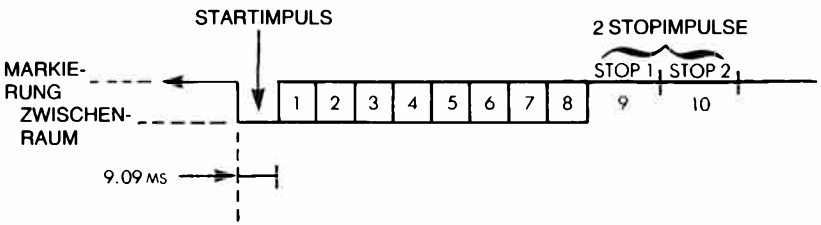


Abb. 6.12: Format eines Fernschreiberwortes

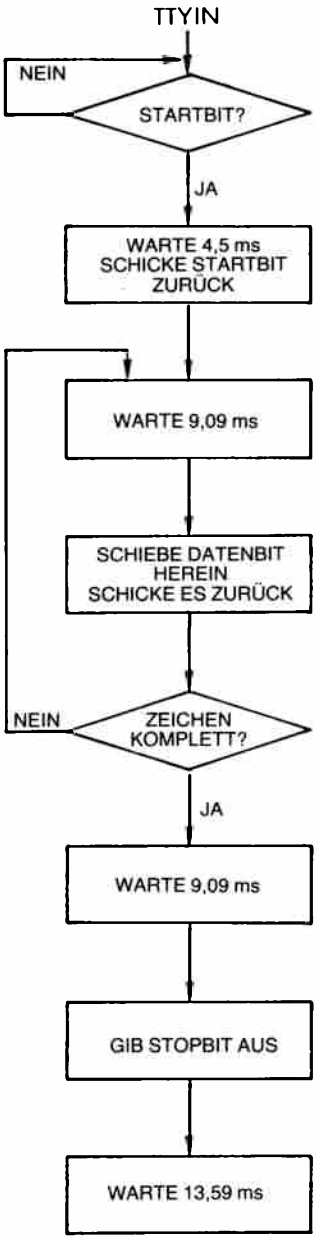


Abb. 6.13: TTY-Eingabe mit Echo

110 Bit pro Sekunde bedeutet, daß die Bits 9,09 Millisekunden auseinander liegen. Solange muß die Verzögerungsschleife dauern, die zwischen aufeinanderfolgende Bits eingebaut wird. Das Format eines Fernschreiberwortes ist in Abb. 6.12 dargestellt. Das Flußdiagramm zur Biteingabe erscheint in Abb. 6.13. Das Programm folgt:

```
TTYIN   IN    A,(STATUS)
        BIT   7,A          Daten bereit?
        JR    Z,TTYIN      Warte sonst
        CALL  DELAY1       Mitte des Impulses
        IN    A,(TTYBIT)   Startbit
        OUT   (TTYBIT),A   Echo
        CALL  DELAY9       Nächster Impuls (9ms)
        LD    B,08H        Bitzähler
NEXT    IN    A,(TTYBIT)   Lies Datenbit
        OUT   (TTYBIT),A   Echo
        SRL   A            Speichere es im Carry
        RR    C            Rotiere es nach C
        CALL  DELAY9       Nächster Impuls (9ms)
        DEC   B            Dekrementiere Bitzähler
        JR    NZ,NEXT
        IN    A,(TTYBIT)   Lies Stopbit
        OUT   (TTYBIT),A   Echo
        CALL  DELAY9       Überspringe zweites Stopbit
        RET
```

Abb. 6.14: Fernschreiberprogramm

Wir wollen das Programm jetzt genau untersuchen. Zuerst muß der Status des Fernschreibers getestet werden, um festzustellen, ob ein Zeichen verfügbar ist:

```
TTYIN   IN    A,(STATUS)
        BIT   7,A
        JR    Z,TTYN
```

Der Befehl „BIT" ist eine nützliche Einrichtung beim Z80, die es erlaubt, jedes Bit in jedem Register zu testen. Dabei wird der Inhalt des getesteten Registers nicht verändert. Das Z-Flag wird gesetzt, wenn der Inhalt des getesteten Registers 0 ist, sonst zurückgesetzt.

Das Programm verbleibt deshalb solange in der Schleife, bis der Status schließlich „1" wird. Dies ist eine klassische Abfrageschleife.

Beachten Sie, daß wir auch

```
        AND   10000000B
```

statt

```
        BIT   7,A
```

verwenden könnten, da der Status nicht erhalten bleiben muß. Der Befehl AND zerstört den Inhalt des Akkumulators (was hier akzeptabel wäre).

Wenn man ein Programm optimiert, dann muß man beachten, daß jeder neue Befehl unterwünschte Nebenwirkungen haben kann.

Danach ist eine Verzögerung von4,5 ms eingebaut, um das Startbit in der Mitte des Impulses abzutasten.

```
        CALL  DELAY1
```

DELAY1 ist ein Unterprogramm, das die gewünschte Verzögerung ergibt. Das erste Bit, das hereinkommen soll, ist das Startbit. Es sollte wieder an den Fernschreiber ausgegeben, ansonsten aber nicht beachtet werden. Dies erledigen die nächsten Befehle:

```
        IN    A,(TTYBIT)
        OUT   (TTYBIT),A
```

Wir müssen dann auf das erste Datenbit warten. Die nötige Verzögerung beträgt 9,09 Millisekunden und ist als Unterprogramm ausgeführt:

```
        CALL  DELAY9
```

Register B wird als Zähler verwendet und wird mit der Zahl 8 geladen, um die 8 Datenbits richtig aufzunehmen:

```
        LD    B,08H
```

Dann werden die Datenbits nacheinander in den Akkumulator gelesen und wieder ausgegeben. Es wird angenommen, daß sie in der Stelle 0 des Akkumulators ankommen. Das Datenbit wird dann ins Register C gerettet, indem es hineingeschoben wird. Der Transfer von A nach C wird über das Carrybit ausgeführt:

```
NEXT    IN    A,(TTYBIT)
        OUT   (TTYBIT),A
        SRL   A
        RR    C
```

Dieser Ablauf ist in Abb. 6.15 veranschaulicht.

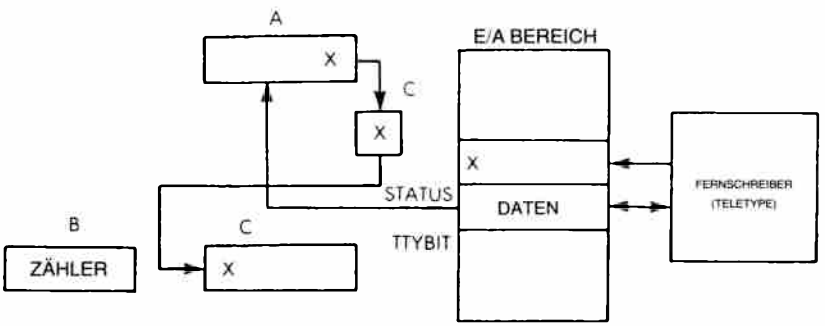


Abb. 6.15: Eingabe vom Fernschreiber

Danach sind die üblichen 9 Millisekunden Verzögerung eingebaut, der Bitzähler wird dekrementiert und die Schleife solange wiederholt, bis die acht Bit aufgenommen sind:

```
CALL  DELAY9
DEC   B
JR    NZ,NEXT
```

Schließlich wird das Stopbit aufgenommen und wieder ausgegeben. Es genügt normalerweise, ein einzelnes Stopbit zu senden, mit zwei weiteren Befehlen könnte man jedoch beide zurückschicken:

```
IN    A,(TTYBIT)
OUT   (TTYBIT),A
CALL  DELAY9
RET
```

Dieses Programm sollte man sorgfältig untersuchen. Der logische Aufbau ist ziemlich einfach. Neu ist, daß jedes Bit, das aufgenommen wird (bei der Adresse TTYBIT), an den Fernschreiber zurückgeschickt wird. Dies ist ein Standardverfahren bei Fernschreibern. Wenn der Benutzer eine Taste drückt, wird die Information an den Prozessor übertragen und dann zum Druckwerk des Fernschreibers zurückgeschickt. Damit wird überprüft, daß die Übertragungsleitungen in Ordnung sind und daß der Prozessor arbeitet, wenn ein Zeichen richtig gedruckt wird.

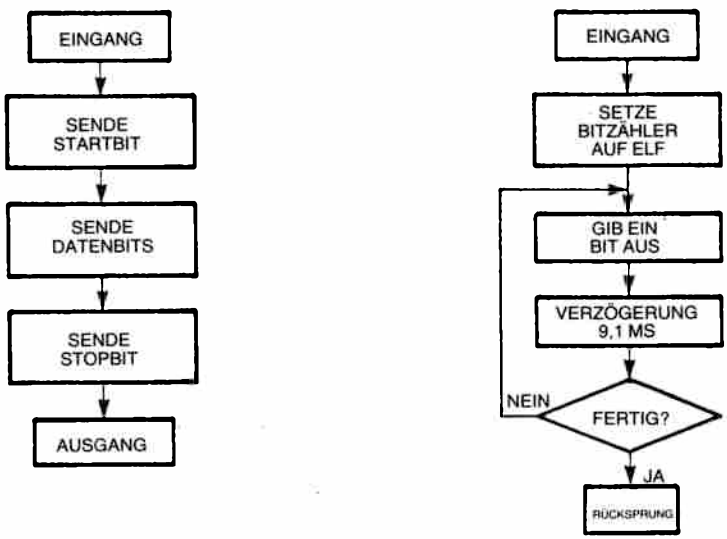


Abb. 6.16: Ausgabe auf den Fernschreiber

Aufgabe 6.25: Schreiben Sie das Verzögerungsprogramm, das die 9,09 Millisekunden Verzögerung liefert. (Unterprogramm DELAY)

Aufgabe 6.26: Schreiben Sie in Anlehnung an das obige Programm ein Programm PRINTC, das den Inhalt der Speicherzelle CHAR auf dem Fernschreiber ausdruckt (siehe Abb. 6.15).

Die Lösung erscheint unten:

```
PRINTC  LD    B,11H          Zähler = 11 Bit
        LD    A,(CHAR)       Hole Zeichen
        OR    A              Lösche Carry = Startbit
        RLA                  Carry nach A
NEXT    OUT   (TTYBIT),A     Ausgabe
        CALL  DELAY
        RRA                  Nächstes Bit
        SCF                  Carry = 1 (Stopbit)
        DEC   B              Bitzähler
        JR    NZ,NEXT
        RET
```

Das Register B dient als Bitzähler für die Ausgabe. Der Inhalt von Bit 0 in A wird an den Fernschreiber geschickt („TTYBIT"). Beachten Sie, daß das Carry verwendet wird, um ein neuntes Bit zu liefern (das Startbit). Beachten Sie auch, daß das Carry gelöscht wird durch:

```
        OR    A
```

Am Ende des Programms wird das Carry auf Eins gesetzt durch:

```
        SCF
```

um das Stopbit zu erzeugen.

Aufgabe 6.27: Verändern Sie das Programm so, daß es auf ein Startbit wartet und nicht auf ein Statusbit.

Ausdrucken einer Zeichenkette

Wir wollen annehmen, daß das Unterprogramm PRINTC (siehe Aufgabe 6.26) ein Zeichen auf unseren Drucker, unsere Anzeige oder auf ein beliebiges Gerät ausgibt. Wir wollen hier den Inhalt des Speichers ab der Stelle START bis zur Stelle START + N ausdrucken.

Das Programm ist einfach (siehe Abb. 6.17):

```
PSTRING LD    B,N            Länge der Zeichenkette
        LD    HL,START       Basisadresse
NEXT    LD    A,(HL)         Hole Zeichen
        CALL  PRINTC         Drucke es
        INC   HL             Nächstes Element
        DEC   B
        JR    NZ,NEXT        Wiederhole den Vorgang
        RET
```

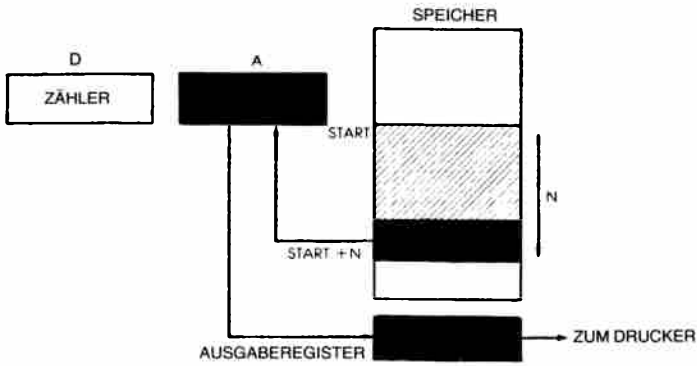


Abb. 6.17: Ausdruck eines Speicherbereichs

Zusammenfassung der Peripherie

Wir haben jetzt die Grundtechniken der Programmierung beschrieben, die man verwendet, um mit typischen Ein-/Ausgabegeräten zu kommunizieren. Zusätzlich zur Übertragung der Daten ist es normalerweise notwendig, ein oder mehrere Steuerregister innerhalb jedes Ein-/Ausgabegerätes zu setzen, um die Übertragungsgeschwindigkeit, den Interruptmechanismus und verschiedene andere Optionen richtig festzulegen. Dazu sollte man das Handbuch für jedes Gerät zu Rate ziehen. (Für weitergehende Einzelheiten über den Informationsaustausch mit allen gebräuchlichen Peripherien sei auf unser Buch *Mikroprozessor Interface Techniken,* Ref.-Nr. 3012 verwiesen.)

Wir haben jetzt gelernt, einzelne Geräte zu verwalten. In einem realen System sind jedoch alle Peripheriegeräte mit Bussen verbunden und können gleichzeitig eine Bearbeitung anfordern. Wie setzen wir dann die Prozessorzeit fest?

Verwaltung von Ein- und Ausgabe

Da Anforderungen von Eingaben und Ausgaben gleichzeitig auftreten können, muß in jedem System ein Verwaltungsmechanismus eingebaut sein, der festlegt, in welcher Reihenfolge die Bearbeitung stattfindet. Es werden drei grundlegende Ein-/Ausgabetechniken angewendet, die miteinander kombiniert werden können. Dies sind: Polling, Interrupt und DMA. Polling und Interrupt werden hier beschrieben. DMA ist eine reine Hardwaretechnik und wird als solche hier nicht beschrieben. (Sie wird in den Büchern 3012 und 3017 abgehandelt.)

Polling

Vom Konzept her ist Polling die einfachste Methode, um mehrere Peripheriegeräte zu verwalten. Bei diesem Verfahren fragt der Prozessor die

Geräte, die an die Busse angeschlossen sind, der Reihe nach ab. Wenn ein Gerät eine Bearbeitung anfordert, wird sie gewährt. Fordert es keine Bearbeitung an, wird das nächste Peripheriegerät untersucht. Polling wird nicht nur für Geräte angewendet, sondern bei *beliebigen Bearbeitungsprogrammen*.

Als Beispiel sei unser System mit einem Fernschreiber, mit einem Tonbandgerät und mit einem Bildschirm ausgestattet. Dann würde das Polling-Programm den Ferschreiber fragen: „Hast du ein Zeichen zu übertragen?" Danach würde es das Fernschreiberausgabeprogramm fragen: „Hast du ein Zeichen zu senden?" Angenommen, die Antworten wären bisher negativ, dann würde es die Tonbandgeräte-Programme und schließlich den Bildschirm abfragen. Ist nur ein Gerät an das System angeschlossen, dann wird Polling auch verwendet, um festzustellen, ob das Gerät nach Bearbeitung verlangt. Als Beispiel sind die Flußdiagramme zum Lesen von einem Lochstreifenleser und zur Ausgabe an einen Drucker in Abb. 6.20 und 6.21 dargestellt.

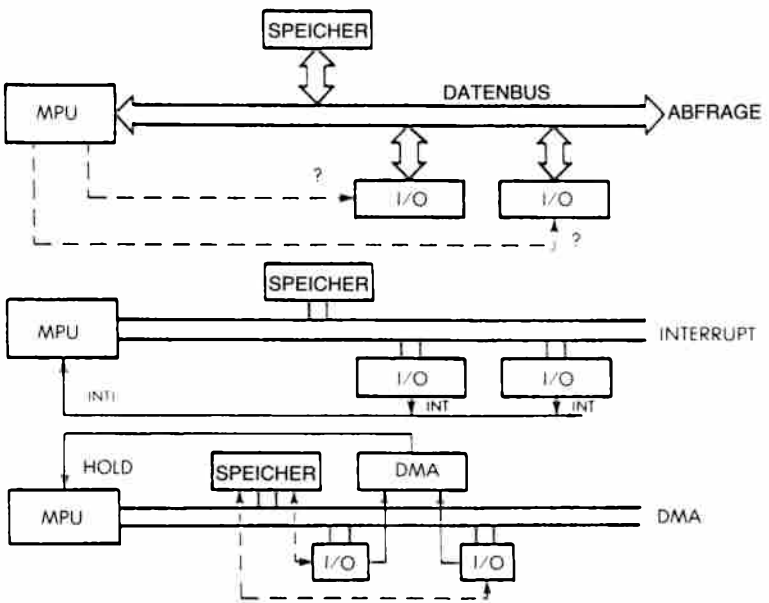


Abb. 6.18: Drei Methoden der Ein-/Ausgabesteuerung

Beispiel: eine Pollingschleife für die Geräte 1, 2, 3 und 4 (siehe Abb. 6.19):

```
POLL4   IN    A,(STATUS1)  Lies Status von Gerät 1
        BIT   7,A          Anforderung einer Bearbeitung?
        CALL  NZ,EINS      BIT 7 = 1?
        IN    A,(STATUS2)  Gerät 2
        BIT   7,A
        CALL  NZ,ZWEI
        IN    A,(STATUS3)  Gerät 3
        BIT   7,A
        CALL  NZ,DREI
        IN    A,(STATUS4)  Gerät 4
        BIT   7,A
        CALL  NZ,VIER
        JR    POLL4        Keine Anforderung, nächster
                           Versuch
```

Bit 7 des Statusregisters von jedem Gerät ist „1", wenn es Bearbeitung verlangt. Wird eine Anforderung erkannt, verzweigt dieses Programm zu dem entsprechenden Bearbeitungsprogramm, bei Adresse EINS für das Gerät 1, ZWEI für das Gerät 2 usw.

Auf eine Feinheit sei hier hingewiesen. Es ist wichtig, sich bei jedem Befehl sorgfältig zu überzeugen, wie er die Flags beeinflußt. Es sollte betont werden, daß der Eingabebefehl die Flags nicht beeinflußt. Deshalb muß mit „BIT 7,A" ein spezieller Test in das Programm eingebaut werden.

Die Vorteile von Polling sind offensichtlich: Das Verfahren ist einfach, benötigt keine Hardwareunterstützung, und alle Ein- und Ausgaben laufen synchron mit dem Programmablauf. Die Nachteile sind genauso offensichtlich: Die meiste Prozessorzeit wird damit verschwendet, nach Geräten zu schauen, die keine Bearbeitung brauchen. Darüberhinaus könnte der Prozessor, da so viel Zeit verschwendet wird, ein Gerät zu spät bearbeiten.

Deshalb wünscht man sich ein anderes Verfahren, um zu gewährleisten, daß die Prozessorzeit für sinnvolle Arbeiten genutzt werden kann und nicht die ganze Zeit nur dazu, unnötig Geräte abzufragen. Wir wollen aber betonen, daß Polling immer dann ausgiebig verwendet wird, wenn ein Mikroprozessor nichts Besseres zu tun hat, da es die gesamte Organisation einfach hält. Wir wollen jetzt die wichtige Alternative zu Polling untersuchen: Interrupts.

Interrupts

Das Konzept von Interrupts ist in Abb. 6.18 vernaschulicht. Eine spezielle Hardwareleitung, die Interruptleitung, ist mit einem speziellen Eingang des Mikroprozessors verbunden. An diese Interruptleitung können mehrere Ein-/Ausgabegeräte angeschlossen werden. Wenn irgendeines davon dann nach Bearbeitung verlangt, legt es einen Pegel oder einen Impuls auf diese Leitung. Ein Interruptsignal ist eine Anfor-

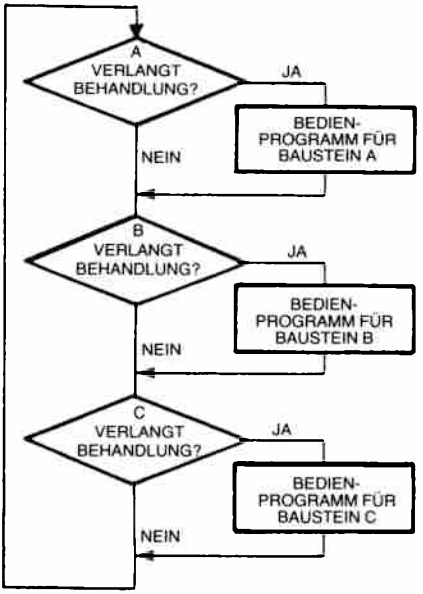


Abb. 6.19: Flußdiagramm einer Pollingschleife

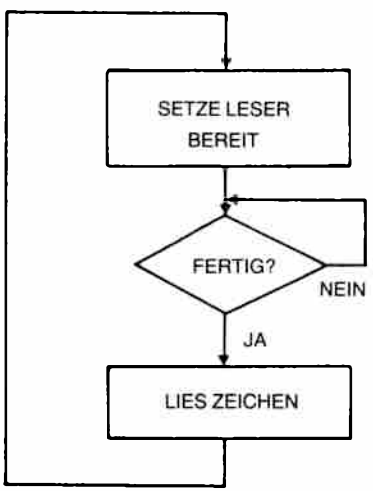


Abb. 6.20: Eingabe von einem Lochstreifenleser

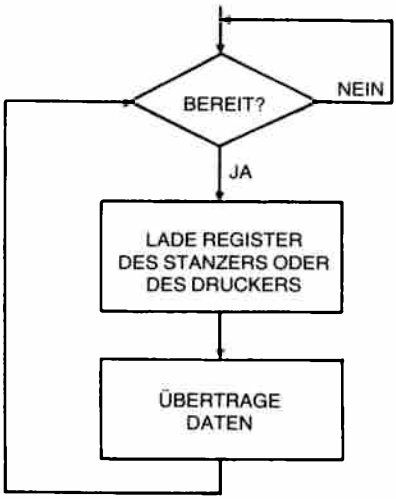


Abb. 6.21: Ausgabe auf einen Stanzer oder Drucker

derung von einem Ein-/Ausgabegerät an den Prozessor. Wir wollen die Antwort des Prozessors auf diesen Interrupt untersuchen.

In jedem Fall arbeitet der Prozessor den Befehl, den er gerade ausführte, vollständig ab. Sonst würde dies ein Chaos im Innern des Mikroprozessors auslösen. Als nächtes sollte der Mikroprozessor zu einem Interruptbehandlungs-Programm verzweigen, das den Interrupt bearbeitet. Die Verzweigung zu einem solchen Unterprogramm setzt voraus, daß der Inhalt des Befehlszählers auf dem Stapel abgelegt werden muß. *Ein Interrupt muß deshalb die automatische Rettung des Befehlszählers auf den Stapel auslösen.* Zusätzlich sollte das Flagregister F automatisch gerettet werden, wenn sein Inhalt von den folgenden Befehlen verändert wird. Wenn die Interruptbehandlungs-Routine irgendwelche internen Register verändert, müssen letztlich auch diese auf den Stapel gerettet werden (siehe Abb. 6.22 und 6.23).

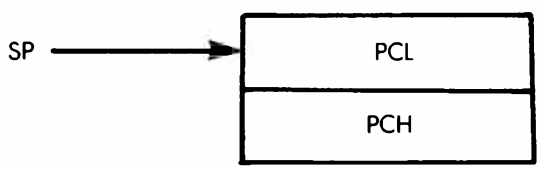


Abb. 6.22: Der Stapel des Z80 nach einem Interrupt

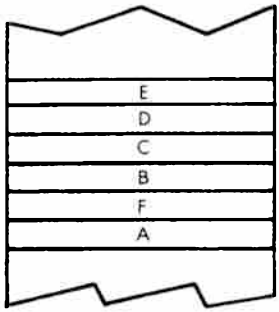


Abb. 6.23: Sicherstellen einiger Arbeitsregister

Nachdem alle diese Register sichergestellt sind, kann man zu der entsprechenden Interruptbehandlungs-Adresse verzweigen. Am Ende dieses Programms sollten alle Register wiederhergestellt werden, und ein spezieller Rücksprung vom Interrupt sollte ausgeführt werden, so daß die Ausführung des Hauptprogramms fortgesetzt wird. Wir wollen die Interruptleitungen des Z80 genauer untersuchen.

Z80 Interrupts

Ein Interrupt ist ein Signal, das zum Mikroprozessor geschickt wird, und das jederzeit asynchron zum Programm Behandlung anfordern kann. Wenn ein Programm zu einem Unterprogramm verzweigt, dann ist eine solche Verzweigung *synchron* zur Programmausführung, d. h. vom Programm gesteuert. Ein Interrupt kann jedoch zu jeder Zeit eintreten und wird im allgemeinen die Ausführung des laufenden Programms unterbrechen (ohne daß das Programm es weiß). Da dies zu beliebiger Zeit im Programm passieren kann, nennt man das *asynchron*.

Der Z80 verfügt über drei Interruptmechanismen: den Bus Request (BUSRQ), den nicht maskierbaren Interrupt (NMI) und den normalen Interrupt (INT).

Wir wollen diese drei Verfahren untersuchen.

Der Bus Request

Der Bus Request ist der Interruptmechanismus mit der höchsten Priorität beim Z80. Abb. 6.24 zeigt den Ablauf. Es ist eine allgemeine Regel, daß der Z80 solange keinen Interrupt zur Kenntnis nimmt, bis der laufende Maschinenzyklus beendet ist. Die Interrupts NMI und INT werden nicht bearbeitet, bis der laufende Befehl abgeschlossen ist. BUSRQ wird jedoch am Ende des laufenden Maschinenzyklus behandelt, ohne daß notwendigerweise das Ende des Befehls abgewartet wird. BUSRQ wird für einen direkten Speicherzugriff verwendet (DMA) und es veranlaßt, daß der Z80 in den DMA-Modus geht (siehe C201 für eine Erklä-

rung des DMA-Mechanismus). Ist das Ende eines Befehls erreicht, und stehen NMI oder INT an, so werden sie im Z80 daran erkannt, daß spezielle Flip-Flops gesetzt werden: das NMI-Flip-Flop und das INT-Flip-Flop. Im DMA Modus unterbricht der Z80 seine Arbeit und gibt seinen Datenbus und seinen Adreßbus im hochohmigen Zustand frei. Diese Betriebsart wird normalerweise von einem DMA-Controller benutzt, um Datenübertragungen zwischen einem sehr schnellen Ein-/Ausgabegerät und dem Speicher durchzuführen und dazu Daten- und Adreßbus des Mikroprozessors zu benutzen. Das Ende der DMA-Operation wird dem Z80 dadurch angezeigt, daß BUSRQ seinen Pegel ändert. An die-

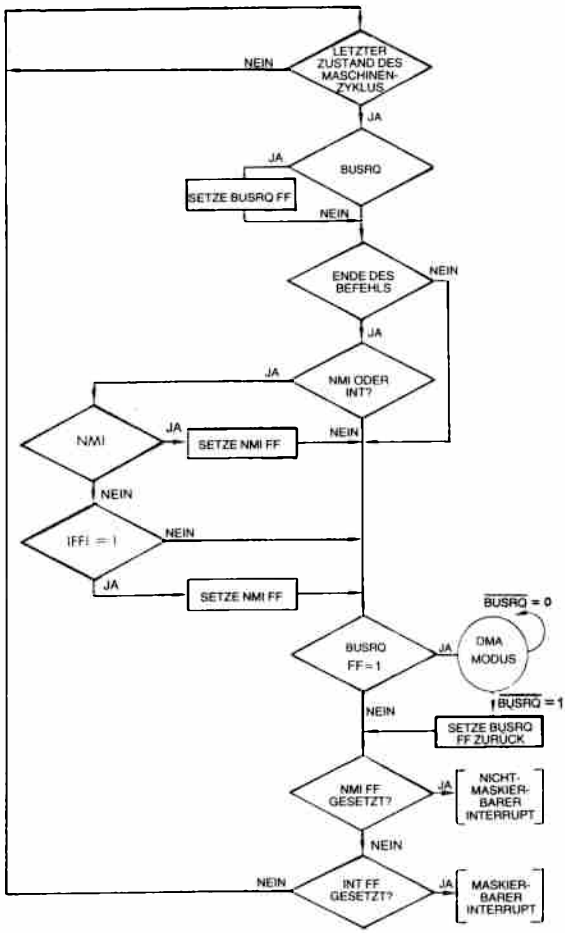


Abb. 6.24: Interrupt Folge

ser Stelle nimmt der Z80 seine normale Arbeit wieder auf. Eventuell testet er zuerst, ob sein internes NMI- oder INT-Flip-Flop gesetzt wurden, und bearbeitet dann die entsprechenden Interrupt.

Normalerweise sollte DMA den Programmierer nicht betreffen, wenn nicht der zeitliche Ablauf wichtig ist. Enthält das System einen DMA-Controller, dann muß der Programmierer verstehen, daß DMA die Antwort auf einen NMI oder INT verzögern kann.

Der Nichtmaskierbare Interrupt

Diese Art von Interrupt kann vom Programmierer nicht unterdrückt werden. Man sagt deshalb, er sei *nicht maskierbar*, daher sein Name. Er wird vom Z80 immer nach Ausführung des laufenden Befehls akzeptiert, falls kein Bus Request empfangen wurde. (Wenn während eines BUSRQ ein NMI empfangen wird, dann wird das interne NMI-Flip-Flop gesetzt, und der Interrupt wird nach dem Bus Request bearbeitet.)

Der NMI bewirkt, daß der Befehlszähler automatisch auf den Stapel abgelegt wird, und daß zur Adresse 0066H verzweigt wird: Die beiden Bytes, die die Adresse 0066H darstellen, werden in den Befehlzähler eingetragen. Sie bilden die Startadresse der Behandlungsroutine für den NMI (siehe Abb. 6.25).

Dieser Interruptmechanismus wurde auf Geschwindigkeit ausgelegt, da er bei „Notfällen" verwendet wird. Deshalb bietet er nicht die Flexibilität der maskierbaren Interruptarten, die unten beschrieben werden.

Beachten Sie auch, daß eine Interruptroutine bei der Adresse 0066H geladen sein muß, bevor der NMI verwendet wird.

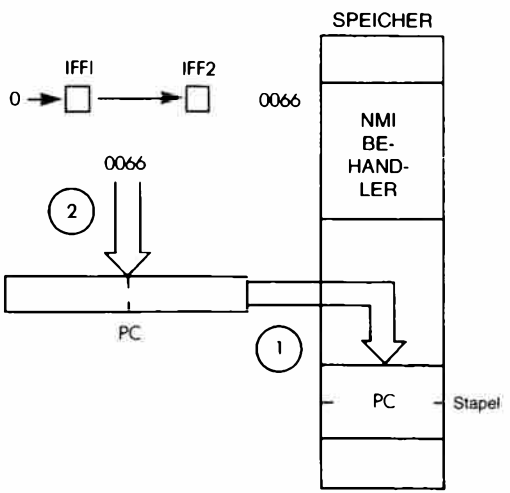


Abb. 6.25: Der NMI erzwingt eine automatische Verzweigung

Ein NMI bewirkt einen automatischen Restart zu der Adresse 0066H. Es laufen die folgenden Ereignisse ab:

```
PC    → Stapel  (rette Befehlszähler)
IFF1  → IFF2    (rette IFF)
0     → IFF1    (setze IFF zurück)
Sprung nach 0066H (starte Interruptbehandlung)
```

Der Zustand des Flip-Flops, das das Bit Interruptmaske enthält (IFF1), wird automatisch nach IFF2 gerettet. Dann wird IFF1 zurückgesetzt um weitere Interrupts zu verhindern. Diese Maßnahme ist wichtig, um zu verhindern, daß Interrupts niedriger Priorität verloren gehen, und sie vereinfacht die externe Hardware: Der Status eines anstehenden INT wird im Z80 intern gespeichert.

Der NMI wird normalerweise für Ereignisse hoher Priorität verwendet, wie für Uhren oder Stromausfälle.

Der Rücksprung von der NMI-Behandlung wird mit einem Spezialbefehl ausgeführt, mit RETN: „Rücksprung von einem Nichtmaskierbaren Interrupt". Der Inhalt von IFF1 wird wieder aus IFF2 geladen und der Inhalt des Befehlszählers vom Stapel geholt. Da IFF1 während der Ausführung des NMI zurückgesetzt war, konnten keine externen INTs akzeptiert werden, d. h. es ging keine Information verloren.

Bei der Beendigung des Interruptbehandlungs-Programms werden folgende Operationen ausgeführt:

```
IFF2  → IFF1  (stelle IFF1 wieder her)
Stapel → PC   (stelle Befehlszähler wieder her)
```

Sobald IFF1 wiederhergestellt ist, liegt wieder der alte Interrupt-Enable-Status vor.

Interrupt

Der normale maskierbare Interrupt INT kann in einem von drei Modi arbeiten. Sie sind spezielle Eigenschaften des Z80, der 8080 hat nur eine einzige Interrupt-Betriebsart. Der normale Interrupt INT kann auch vom Programmierer selektiv maskiert werden. Wenn die Interrupt-Flip-Flops IFF1 und IFF2 auf „1" gesetzt werden, dann sind Interrupts zugelassen. Werden sie auf „0" gesetzt (maskiert), wird INT nicht beachtet. Mit dem Befehl EI werden sie gesetzt, mit DI zurückgesetzt. IFF1 und IFF2 werden zusammen gesetzt oder zurückgesetzt. Während der Ausführung von EI und DI sind INTs gesperrt, um einen Verlust von Information zu vermeiden.

Wir wollen jetzt die drei Interrupt-Modi untersuchen:

Interrupt Modus 0

Dieser Modus ist mit dem 8080 Interrupt Modus identisch. Der Z80 arbeitet im Interrupt-Modus 0, nachdem er anfangs gestartet wurde (wenn das Reset-Signal angelegt wurde) oder wenn der Befehl IM0 ausgeführt wurde. Ist der Interrupt-Modus 0 gesetzt, dann wird ein Interrupt er-

kannt, wenn das Interrupt-Freigabe-Flip-Flop IFF1 auf 1 gesetzt ist, vorausgesetzt daß nicht gleichzeitig ein Bus Request oder ein nichtmaskierbarer Interrupt auftritt. Der Interrupt wird nur am Ende eines Befehls erkannt, Im wesentlichen antwortet der Z80 auf einen Interrupt, indem er das Signal IORQ erzeugt (außer dem Signal M1) und dann wartet.

Es ist die Aufgabe eines *externen Gerätes*, die Signlae IORQ und M1 (dies nennt man *Interrupt Acknowledge* oder INTA) zu erkennen, und einen Befehl auf den Datenbus zu legen. Der Z80 erwartet, daß ein externes Gerät während des nächsten Zyklus einen Befehl auf den Datenbus legt. Typischerweise wird ein RST oder ein CALL auf den Bus gelegt. Diese beiden Befehle retten automatisch den Befehlzähler auf den Stapel und bewirken eine Verzweigung zu einer speziellen Adresse. Der Vorteil des Befehls RST ist es, daß er nur aus einem einzigen Byte besteht, d. h. daß er schnell ausgeführt wird. Sein Nachteil ist, daß er nur zu einer von acht Adressen in der Seite Null (Adressen 0 bis 255) verzweigen kann. Der Vorteil des Befehs CALL ist es, daß er als universeller Verzweigungsbefehl eine vollständige 16-Bit-Adresse festlegt. Er belegt jedoch drei Byte und wird deshalb weniger schnell ausgeführt.

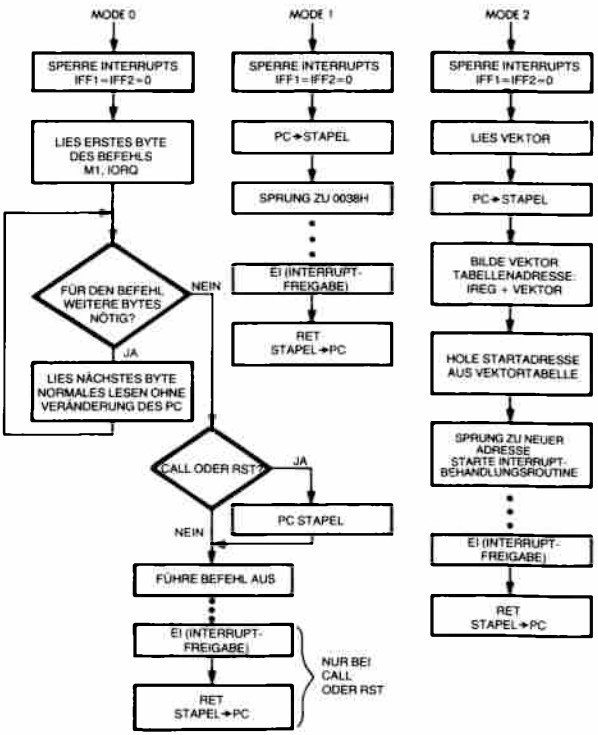


Abb. 6.26: Interrupt Modi

Beachten Sie, daß alle weiteren Interrupts gesperrt sind, sobald die Interruptbehandlung beginnt. IFF1 und IFF2 werden automatisch auf „0" gesetzt. Es ist dann Aufgabe des Programmierers, einen Befehl EI (Enable Interrupt, Interruptfreigabe) an der entsprechenden Stelle im Programm einzufügen, wenn er will, daß weitere Interrupts zugelassen sind, und in jedem Fall, bevor er von dem Interrupt zurückkehrt.

Die genaue Abfolge, die dem Modus 0 entspricht, ist in Abb. 6.26 dargestellt.

Der Rücksprung vom Interrupt wird mit dem Befehl RETI ausgeführt. Wir wollen den Programmierer an dieser Stelle daran erinnern, daß er normalerweise dafür verantwortlich ist, daß die Interrupt-Anforderung des Ein-/Ausgabegerätes explizit gelöscht wird, und immer dafür, daß das interne Interrupt-Flip-Flop des Z80 wieder gesetzt wird. Das periphere Steuergerät kann jedoch das Signal INTA dazu verwenden, die Interrupt-Anforderung zu löschen, und den Programmierer von dieser Aufgabe zu entlasten.

Falls die Interruptbehandlungs-Routine den Inhalt irgendwelcher interner Register verändert, ist der Programmierer speziell dafür verantwortlich, diese Register vor Ausführung der Interruptbehandlungs-Routine auf den Stapel zu retten. Sonst wird der Inhalt dieser Register zerstört, und das unterbrochene Programm versagt, wenn es die Ausführung fortsetzen soll. Werden beispielsweise die Register A, B, C, D, E, H und L innerhalb der Interruptbehandlung verwendet, müssen sie gerettet werden (siehe Abb. 6.27).

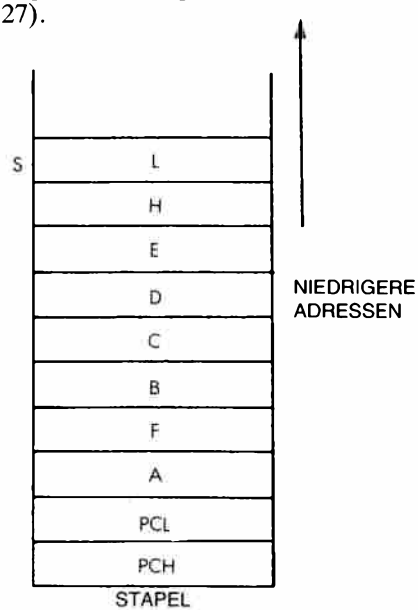


Abb. 6.27: Sicherstellen der Register

Das entsprechende Programm lautet:

```
SAVREG  PUSH  AF
        PUSH  BC
        PUSH  DE
        PUSH  HL
```

Nach Beendigung der Interruptbehandlungs-Routine müssen diese Register wiederhergestellt werden. Das Interruptbehandlungs-Programm wird mit den folgenden Befehlen abgeschlossen:

```
POP   HL
POP   DE
POP   BC
POP   AF
EI          (wenn EI nicht schon vorher
             in diesem
             Programm ausgeführt wurde)
```

Wenn die Routine die Register IX und IY benutzt, müssen diese zusätzlich genauso sichergestellt und später wiederhergestellt werden.

Interrupt Modus 1

Dieser Interrupt Modus wird gesetzt durch Ausführung des Befehls IM1. Er ist ein automatischer Interrupt-Behandler, der einen Sprung zur Adresse 0038H bewirkt. Er ist deshalb im wesentlichen analog zu dem NMI Interruptmechanismus, außer daß er maskiert werden kann. Der Z80 rettet automatisch den Inhalt des Befehlszählers auf den Stapel (siehe Abb. 6.28).

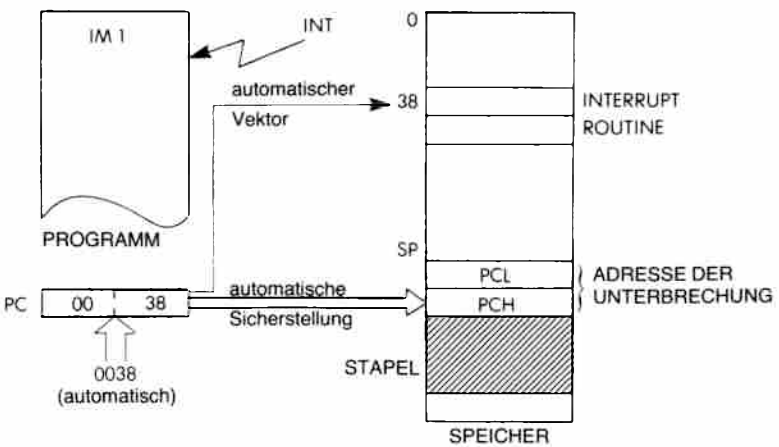


Abb. 6.28: Interrupt Modus 1

Diese automatische Antwort auf einen Interrupt, die alle Interrupts auf die Adresse 38H verweist, vermindert den Aufwand an externer Hardware, die zur Ausführung von Interrupts nötig ist. Ein möglicher Nachteil ist die Verzweigung zu einer *einzigen* Speicheradresse. Wenn mehrere Bausteine an die Leitung INT angeschlossen sind, dann ist es Aufgabe des Programms an der Adresse 38H festzustellen, welches Gerät den Interrupt angefordert hat. Dieses Problem wird unten angesprochen.

Eine Vorsichtsmaßnahme muß mit Rücksicht auf den zeitlichen Ablauf dieses Interrupts ergriffen werden: Wenn man programmierte Ein-/Ausgaben ausführt, dann beachtet der Z80 Daten nicht, die während des Zuklus, der auf den Interrupt folgt (dem Interrupt Acknowledge Zyklus), auf dem Datenbus liegen können.

Interrupt Modus 2 (Vektorinterrupts)

Dieser Modus wird durch Ausführung des Befehls IM2 gesetzt. Er ist ein leistungsfähiger Modus, der eine automatische Verzeigerung von Interrupts erlaubt. Der Interruptvektor ist eine Adresse, die das periphere Gerät liefert, das den Interrupt angefordert hat, und die als Zeiger auf die Interruptbehandlungsroutine dient. Der Adressierungsmechanismus des Z80 im Modus 2 ist indirekt (statt direkt). Jedes periphere Gerät liefert eine 7 Bit lange Adresse, die an die 8 Bit lange Adresse im Register I angehängt wird. Das rechte Bit der endgültigen 16-Bit-Adresse wird auf Null gesetzt. Diese Adresse zeigt in eine Tabelle irgendwo im Speicher. Diese Tabelle kann bis zu 128 Eintragungen aus zwei Worten enthalten. Jedes dieser Doppelworte ist die Adresse der Behandlungsroutine für das entsprechende Gerät. Dies ist in Abbildung 6.29 und 6.30 veranschaulicht.

Die Interrupt-Tabelle kann bis zu 128 Eintragungen aus zwei Worten enthalten.

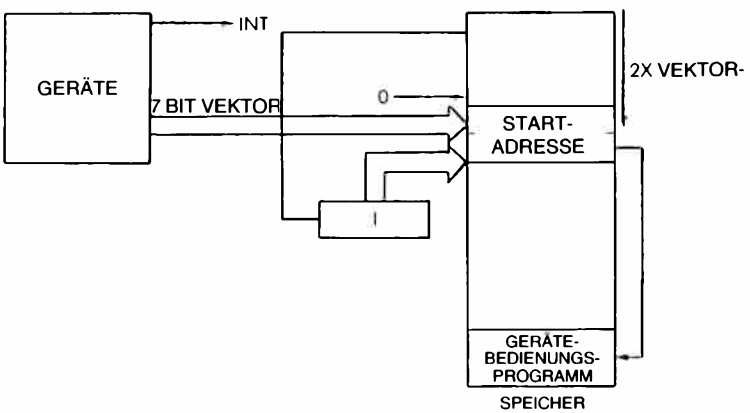


Abb. 6.29: Interrupt Modus 2

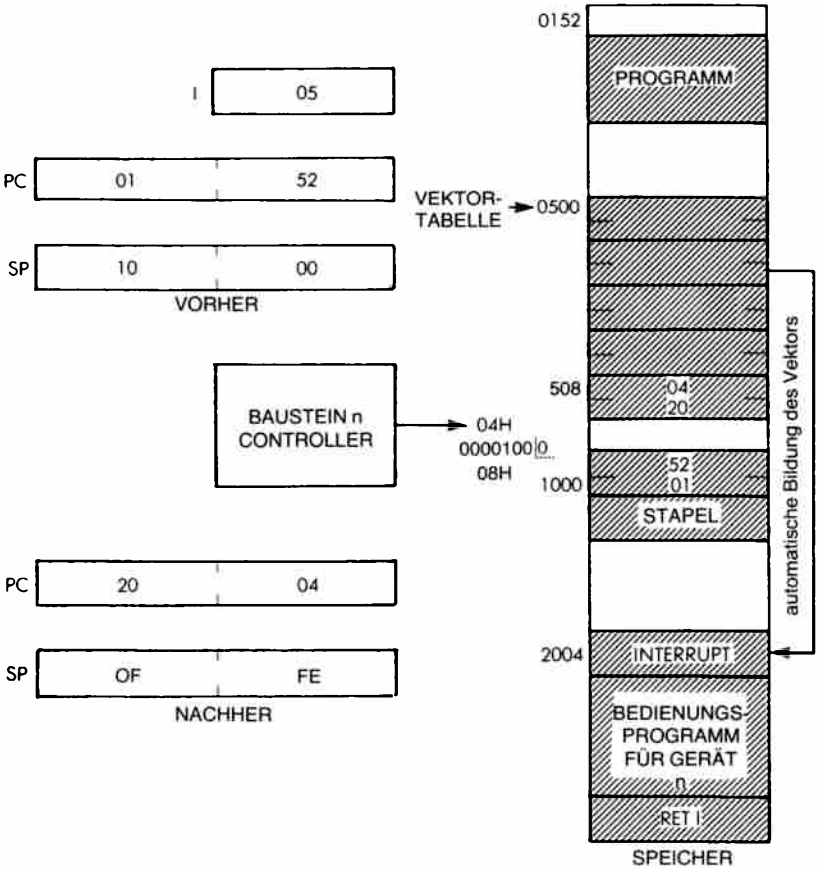


Abb. 6.30: Modus 2 – ein praktisches Beispiel

Auch in diesem Modus legt der Z80 den Inhalt des Befehlszählers automatisch auf den Stapel. Dies ist auch notwendig, da der Befehlszähler aus der Interrupt-Tabelle entsprechend dem Zeiger geladen wird, den das Gerät liefert.

Verwaltungsaufwand bei Interrupts

Für einen Vergleich zwischen Polling und Interrupt sei auf Abb. 6.18 verwiesen, wo oben der Pollingprozeß und darunter der Interruptprozeß dargestellt ist. Man sieht, daß das Programm bei der Pollingtechnik viel Zeit beim abtasten verschwendet.

Wenn man Interrupts verwendet, wird das Programm unterbrochen, der Interrupt behandelt und dann die Ausführung des Programms fortge-

setzt. Der offensichtliche Nachteil eines Interrupts ist es jedoch, daß am Anfang und am Ende mehrere zusätzliche Befehle ausgeführt werden, die eine Verzögerung bewirken, bevor der erste Befehl des Behandlungsprogramms ausgeführt werden kann. Dies ist ein zusätzlicher Zeitverlust.

Aufgabe 6.28: Verwenden Sie die Tabelle im Anhang, die die Zahl der Zyklen für jeden Befehl angibt, und berechnen Sie, wieviel Zeit verloren geht, um die Register A, B, D und H zu retten und wiederherzustellen.

Nachdem die Arbeitsweise der Interruptleitungen geklärt ist, wollen wir uns noch mit zwei weiteren wichtigen Problemen befassen:

1 – Wie lösen wir das Problem, daß mehrere Geräte gleichzeitig einen Interrupt auslösen können?

2 – Wie lösen wir das Problem, daß ein weiterer Interrupt eintreten kann, während ein Interrupt bearbeitet wird?

Verschiedene Geräte an einer einzigen Interruptleitung

Immer wenn ein Interrupt auftritt, verzweigt der Prozessor zu einer festgelegten Adresse. Bevor die Bearbeitung beginnen kann, muß das Interruptbehandlungs-Programm feststellen, welches Gerät den Interrupt ausgelöst hat. Es gibt wie üblich zwei Methoden, das Gerät zu identifizieren: eine Software-Methode und eine Hardware-Methode.

Bei der Software-Methode wird Polling angewendet: Der Mikroprozessor fragt jedes Gerät der Reihe nach: „Hast du den Interrupt angefordert?" Ist die Antwort negativ, wird das nächste Gerät befragt. Dieses Verfahren ist in Abb. 6.31 dargestellt. Ein Beispielprogramm ist:

```
POLINT  IN    A,(STATUS1)   Lies Status
        BIT   7,A           Hat dieses Gerät den INT
                            angefordert?
        JP    NZ,EINS       Wenn ja, behandle den Interrupt
        IN    A,(STATUS2)
        BIT   7,A
        JP    NZ,ZWEI
        usw.
```

Die Hardware-Methode benötigt zusätzliche Bauteile, liefert die Adresse des unterbrechenden Gerätes aber gleichzeitig mit der Interruptanforderung. Der Baustein, der jetzt allgemein verwendet wird, um diese Aufgabe zu erfüllen, heißt PIC oder Priority-Interupt-Controller (priorisierter Steuerbaustein für Interrupts). Ein solcher PIC legt die Adresse automatisch auf den Datenbus, die für das unterbrechende Gerät gebraucht wird.

Um genauer zu sein, im Interrupt Modus 0 legt der PIC einen Ein-Byte-RST oder einen Drei-Byte-CALL auf den Datenbus, als Antwort auf

den Interrupt Acknowledge. So wird der Zeiger auf die Behandlungsroutine automatisch geliefert und der Zeitverlust klein gehalten.

Beachten Sie, daß ein Unterprogrammaufruf nötig ist, da der Z80 im Modus 0 den Befehlszähler nicht rettet.

In den meisten Fällen ist die Geschwindigkeit der Reaktion auf einen Interrupt nicht kritisch, und man verwendet ein Polling-Verfahren. Ist die Antwortzeit besonders entscheidend, muß man ein Hardware-Verfahren anwenden.

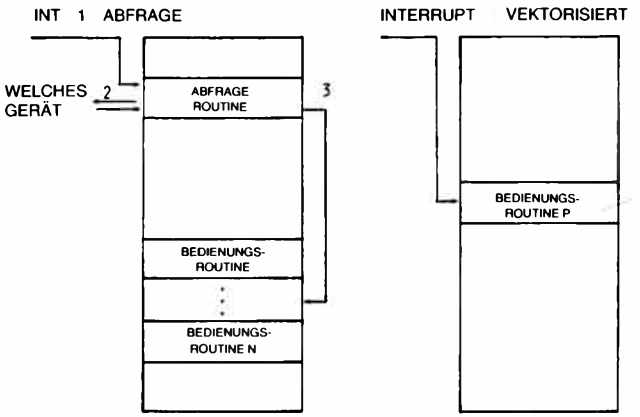


Abb. 6.31: Abfrage (Polling) und vektorisierter Interrupt

Gleichzeitige Interrupts

Weiterhin kann das Problem auftreten, daß ein neuer Interrupt angefordert wird, während ein Interruptbehandlungs-Programm läuft. Wir wollen untersuchen, was dann passiert, und wie der Stapel verwendet wird, um das Problem zu lösen. Im Kapitel 2 haben wir angekündigt, daß dies eine weitere wesentliche Aufgabe des Stapels ist, und jetzt ist die Zeit gekommen, seine Verwendung zu demonstrieren. Wir wollen uns auf Abb. 6.33 beziehen, um mehrfache Interrupts zu veranschaulichen. In der Darstellung verläuft die Zeitachse von links nach rechts. Der Inhalt des Stapels ist im unteren Teil der Abbildung dargestellt. Wenn wir links beginnen, so wird zum Zeitpunkt T0 das Programm P ausgeführt. Wenn wir uns nach rechts wenden, dann tritt zum Zeitpunkt T1 der Interrupt I1 ein. Wir wollen annehmen, daß Interrupts freigegeben sind und I1 zugelassen ist. Das Programm P wird unterbrochen. Dies ist im unteren Teil der Abbildung dargestellt. Der Stapel enthält den Befehlszähler und wenigstens das Statusregister vom Programm P1, außerdem weitere Register, die die Behandlungsroutine für den Interrupt I1 rettet.

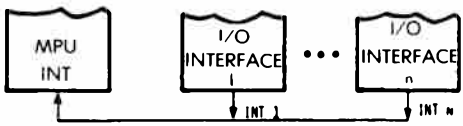


Abb. 6.32: Mehrere Geräte können die gleiche Interruptleitung verwenden

Vom Zeitpunkt T1 bis zum Zeitpunkt T2 wird der Interrupt I1 behandelt. Zum Zeitpunkt T2 tritt der Interrupt I2 auf. Wir wollen annehmen, daß der Interrupt I2 eine höhere Priorität besitzt als I1. Hätte er eine niedrigere Priorität, dann würde er solange nicht beachtet, bis I1 abgeschlossen wäre. Zum Zeitpunkt T2 werden die Register von I1 auf den Stapel gelegt, und dies erscheint unten in der Abbildung. Wieder werden der Inhalt des Befehlszählers und AF auf den Stapel gelegt. Zusätzlich wird vielleicht das Programm für I2 einige weitere Register retten. I2 wird jetzt ausgeführt, bis es zum Zeitpunkt T3 beendet ist.

Wird I2 beendet, dann wird der Inhalt des Stapels automatisch in den Z80 zurück „gepopt", und dies zeigt Abb. 6.33 unten. Dann fährt automatisch I1 mit der Ausführung fort. Unglücklicherweise passiert zum Zeitpunkt T4 wieder ein Interrupt mit höherer Priorität. Wir sehen in der Abbildung, daß wieder die Register von I1 auf den Stapel abgelegt werden. Der Interrupt I3 wird von T4 bis T5 bearbeitet, wo er beendet wird. Zu diesem Zeitpunkt wird der Inhalt des Stapels in den Z80 zurückgegeben, und der Interrupt I1 wird weiter behandelt, bis er zum Zeitpunkt T6 beendet wird. Bei T6 werden die Register, die noch auf dem Stapel liegen, in den Z80 zurückgegeben, und die Ausführung des Programms P wird fortgesetzt. Der Leser wird sich davon überzeugen, daß der Stapel an dieser Stelle leer ist. Die Zahl der gestrichelten Linien, die anzeigt, wie oft ein Programm unterbrochen wurde, zeigt gleichzeitig an, wieviele Ebenen auf dem Stapel liegen.

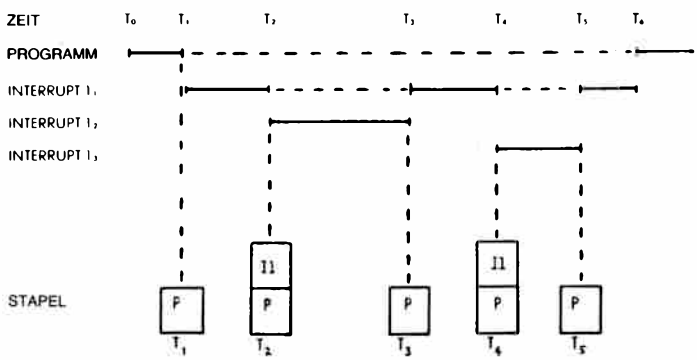


Abb. 6.33: Inhalt des Stapels während mehrerer Interrupts

Aufgabe 6.29: Nehmen sie an, daß der Platz für den Stapel in einem speziellen Programm auf 300 Plätze begrenzt ist. Nehmen Sie an, daß immer alle Register gerettet werden müssen, und daß der Programmierer verschachtelte Interrupts zuläßt, d. h. sie unterbrechen sich gegenseitig. Wieviele Interrupts können gleichzeitig behandelt werden? Kann ein anderer Grund die maximale Zahl gleichzeitiger Interrupts weiter vermindern?

Es muß jedoch betont werden, daß Mikroprozessorsysteme in der Praxis normalerweise mit einer kleinen Zahl von Geräten verbunden sind, die Interrupts auslösen können. Es ist deshalb untypisch, daß eine große Zahl von Interrupts in einem solchen System gleichzeitig auftritt.

Wir haben jetzt alle die Probleme gelöst, die üblicherweise im Zusammenhang mit Interrupts auftreten. Ihre Anwendung ist tatsächlich einfach, und auch der Programmiererneuling sollte sie zum Vorteil einsetzen.

Zusammenfassung

In diesem Kapitel haben wir den Bereich der Techniken vorgestellt, die zur Kommunikation mit der Außenwelt verwendet werden. Von elementaren Ein-/Ausgabeprogrammen bis zu komplexeren Programmen zur Kommunikation mit wirklichen Geräten haben wir es gelernt, die nötigen Programme zu entwickeln, und wir haben sogar die Effizienz eines Benchmarkprogramms im Falle einer Parallelübertragung und einer Parallel-Seriell-Wandlung untersucht. Schließlich haben wir es gelernt, die Arbeit mit mehreren Peripheriegeräten zu verwalten und dabei Polling und Interrupts zu verwenden. Natürlich kann man viele weitere exotische Ein-/Ausgabegeräte an ein System anschließen. Mit den Techniken, die bisher vorgestellt wurden, und mit einem Verständnis für die verwendeten Geräte sollte es möglich sein, die meisten allgemeinen Probleme zu lösen.

Im nächsten Kapitel wollen wir die wirklichen Eigenschaften der Ein-/Ausgabegeräte untersuchen, die normalerweise an einen Z80 angeschlossen werden. Dann wollen wir die grundlegenden Datenstrukturen betrachten, die der Programmierer verwenden kann.

Aufgabe 6.30: Berechnen sie den Zeitverlust, wenn man im Modus 0 arbeitet und alle Register gerettet werden müssen, und wenn ein RST als Antwort auf den Interrupt Acknowledge empfangen wird. Der Zeitverlust ist die gesamte Verzögerung, die eintritt, mit Ausnahme der Befehle, die nötig sind, um die Interruptbearbeitung entsprechend einzubauen.

Aufgabe 6.31: Eine 7-Segment-LED-Anzeige kann auch andere Zeichen als das Hexadezimale Alphabet anzeigen. Berechnen Sie den Kode für H, I, J, L, O, P., S, U, Y, g i, j, l, n, o, p, r, t, u, y.

Aufgabe 6.32: Das Flußdiagramm zur Interruptverwaltung erscheint in Abb. 6.34. Beantworten Sie folgende Fragen:

a – Was erledigt man mit Hardware, was mit Software?
b – Wozu wird die Maske verwendet?
c – Wie viele Register sollte man retten?
d – Wie erkennt man das unterbrechende Gerät
e – Was macht der Befehl RETI? Wie unterscheidet er sich von einem Rücksprung aus einem Unterprogramm?
f – Schlagen Sie eine Methode vor, wie man einen Überlauf des Stapels behandeln kann.
g – Wie groß ist der Zeitverlust, den der Interruptmechanismus mit sich bringt?

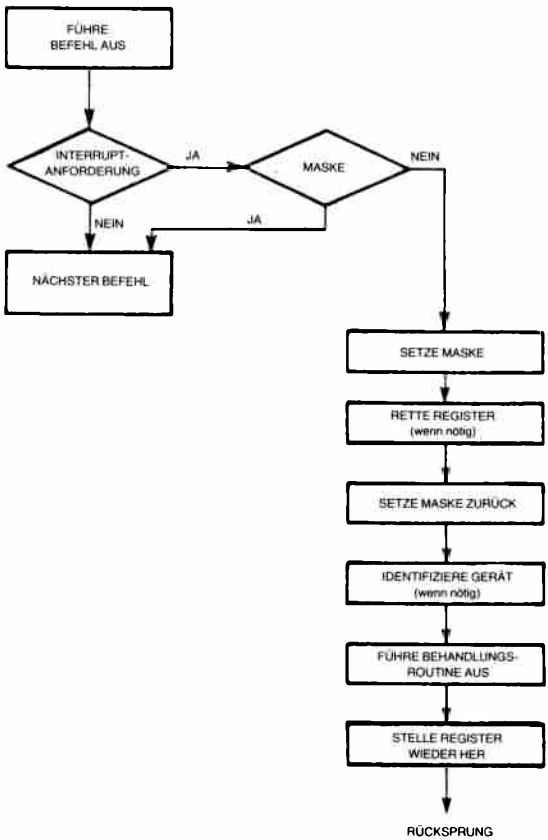


Abb. 6.34: Interruptlogik

7

Ein-/Ausgabegeräte

Einführung

Wir haben gelernt, wie der Mikroprozessor Z80 in den meisten gebräuchlichen Situationen programmiert wird. Wir sollten jedoch den Ein-/Ausgabebausteinen, die normalerweise an den Mikroprozessor angeschlossen werden, besondere Beachtung schenken. Durch den Fortschritt bei der LSI-Integration wurden neue ICs eingeführt, die es vorher nicht gab. Deshalb gehört zur Programmierung eines Systems natürlich die Programmierung des Mikroprozessors selbst, aber auch die *Programmierung der Ein-/Ausgabebausteine.* Tatsächlich ist es oft schwieriger, sich zu merken, wie die verschiedenen Steuerungsvarianten eines Ein-/Ausgabebausteins programmiert werden; wie der Mikroprozessor selbst programmiert wird, ist dagegen vergleichsweise einfach! Dies liegt nicht daran, daß die Programmierung selbst komplizierter ist, sondern daran, daß jeder dieser Bausteine seine speziellen Eigenheiten hat. Wir wollen hier zuerst den allgemeinsten Ein-/Ausgabebaustein untersuchen, den programmierbaren Ein-/Ausgabebaustein (englisch: parallel input output, kurz „PIO"), danach einige Ein-/Ausgabebausteine von Zilog.

Die „Standard-PIO".

Es gibt keine „Standard-PIO". Allerdings funktioniert jeder PIO-Baustein im wesentlichen analog zu allen ähnlichen PIOs, die andere Hersteller für die gleichen Aufgaben produzieren. Die Aufgabe einer PIO ist es, mehrere Ports zur Ein-/Ausgabe zur Verfügung zu stellen (ein „Port" ist einfach ein Satz von 8 Ein-/Ausgabe-Leitungen). Jede PIO stellt wenigstens zwei Sätze von 8 Ein-/Ausgabe-Leitungen zur Ein-/Ausgabe zur Verfügung. Jede PIO braucht einen *Datenpuffer*, um wenigstens den Inhalt des Datenbusses am Ausgang zu stabilisieren. Unsere PIO hat deshalb für jeden Port mindestens einen Puffer.

Zusätzlich haben wir festgesetzt, daß der Mikroprozessor ein *Handshaking-Verfahren* benutzt oder aber mit dem Ein-/Ausgabebaustein über *Interrupts* kommuniziert. Die PIO selbst wird ein ähnliches Verfahren

zur Kommunikation mit der Peripherie anwenden. Jede PIO muß deshalb für jeden Port mindestens zwei *Steuer-Leitungen* besitzen, um das Handshaking-Verfahren zu realisieren.

Der Mikroprozessor muß außerdem den Status jedes Ports lesen können. Jeder Port muß ein oder mehrere Statusbits besitzen. Schließlich wird es in der PIO eine Reihe von Optionen geben, die ihre Funktionen festlegen. Der Programmierer muß auf ein Spezialregister in der PIO zugreifen können, um die programmierbaren Optionen festzulegen. Dies ist das *Steuer-Register*. In manchen Fällen ist die Statusinformation Teil des Steuer-Registers.

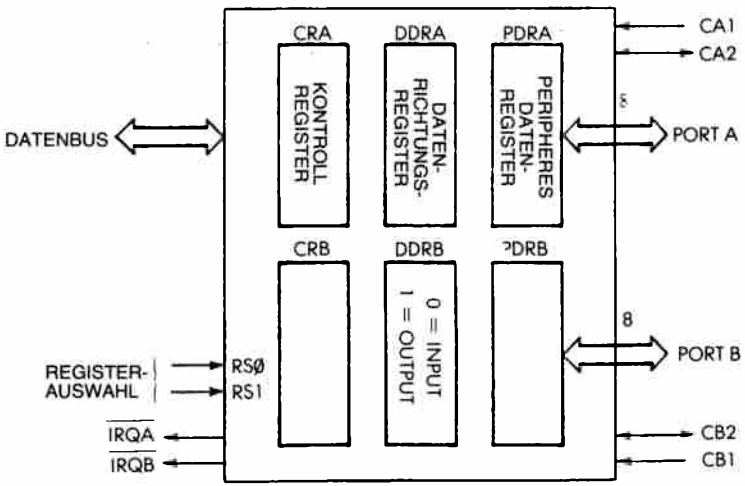


Abb. 7.1: Typische PIO

Eine wesentliche Eigenschaft der PIO ist es, daß jeder Anschluß entweder als Eingang oder als Ausgang programmiert werden kann. Abb. 7.1 zeigt den Aufbau einer PIO. Der Programmierer kann festlegen, ob eine Leitung Eingang oder Ausgang ist. Um die Richtung der Leitungen festzulegen, gibt es ein Datenrichtungsregister für jeden Port. Eine „0" an einer Stelle des Datenrichtungsregisters legt einen Eingang fest, eine „1" einen Ausgang.

Es mag überraschen, daß eine „0" für Eingänge verwendet wird und eine „1" für Ausgänge, und daß nicht umgekehrt eine „0" Ausgänge festlegt und eine „1" Eingänge. Dies ist aber beabsichtigt: Wenn das System eingeschaltet wird, dann ist es wichtig, daß alle Ein-/Ausgänge als Eingänge geschaltet sind. Wenn der Mikrocomputer mit „gefährlichen" Geräten

verbunden ist, könnte er sie sonst durch Zufall einschalten. Wird ein Reset angelegt, dann werden normalerweise alle Register auf Null gesetzt, und das führt dazu, daß alle Anschlüsse der PIO zu Eingängen werden. Die Verbindungen zum Mikroprozessor erscheinen auf der linken Seite der Abbildung. Die PIO ist natürlich an den 8-Bit Datenbus angeschlossen, an den Adreßbus des Mikroprozessors und an den Steuerbus. Der Programmierer gibt einfach die Adresse eines beliebigen Registers an, auf das er innerhalb der PIO zugreifen will.

Die internen Steuer-Register

Das Steuer-Register der PIO liefert eine Anzahl von Optionen zur Erzeugung und Erkennung von Interrupts und zum Einbau automatischer Handshaking-Funktionen. Es ist hier nicht notwendig, alle Möglichkeiten vollständig zu beschreiben. Der Benutzer eines wirklichen Systems, das eine PIO verwendet, muß einfach das Datenblatt zu Rate ziehen, das die Wirkung der verschiedenen Bits im Steuer-Register beschreibt. Wenn das System initialisiert wird, muß der Programmierer das Steuer-Register der PIO mit dem richtigen Inhalt für die gewünschte Anwendung laden.

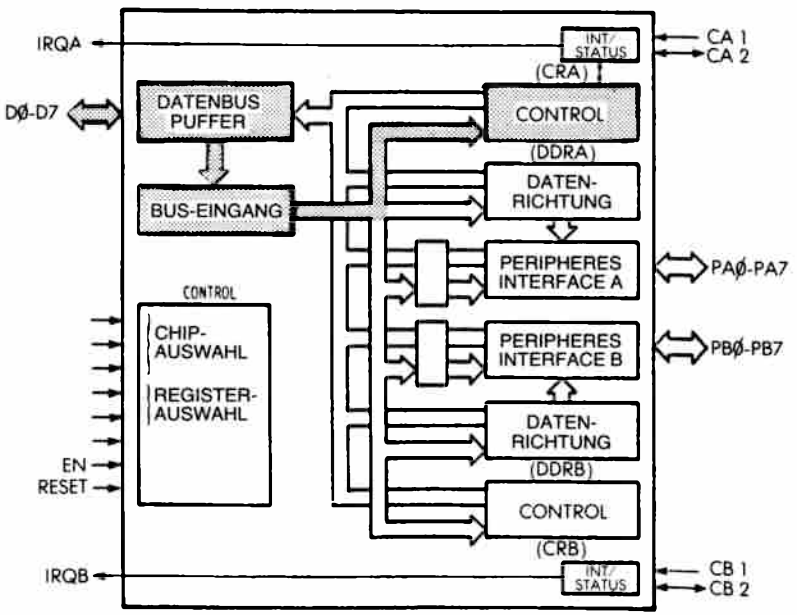


Abb. 7.2: Der Gebrauch einer PIO – Laden des Steuer-Registers

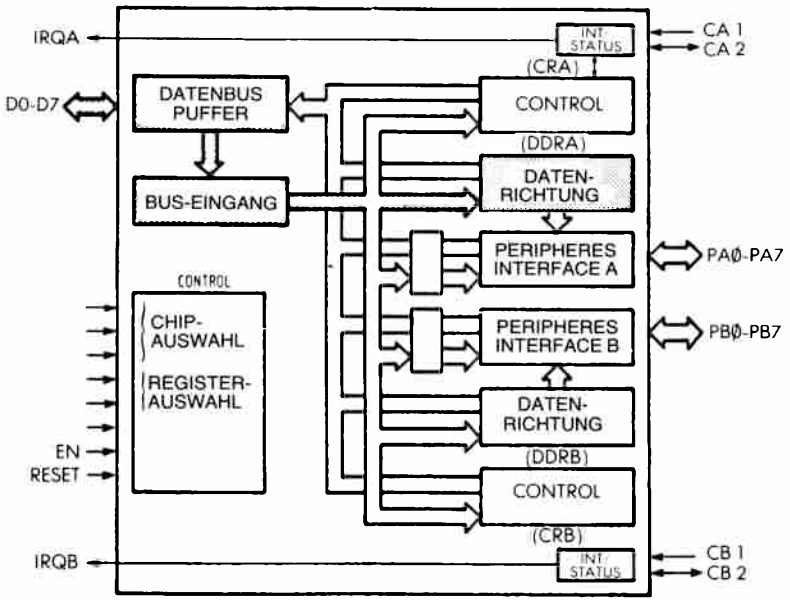


Abb. 7.3: Der Gebrauch einer PIO – Laden der Datenrichtung

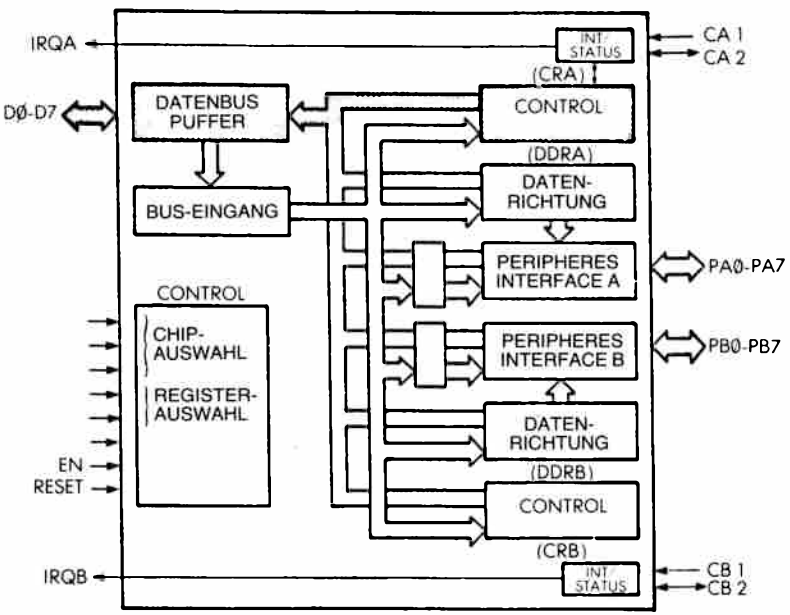


Abb. 7.4: Der Gebrauch einer PIO – Lesen des Status

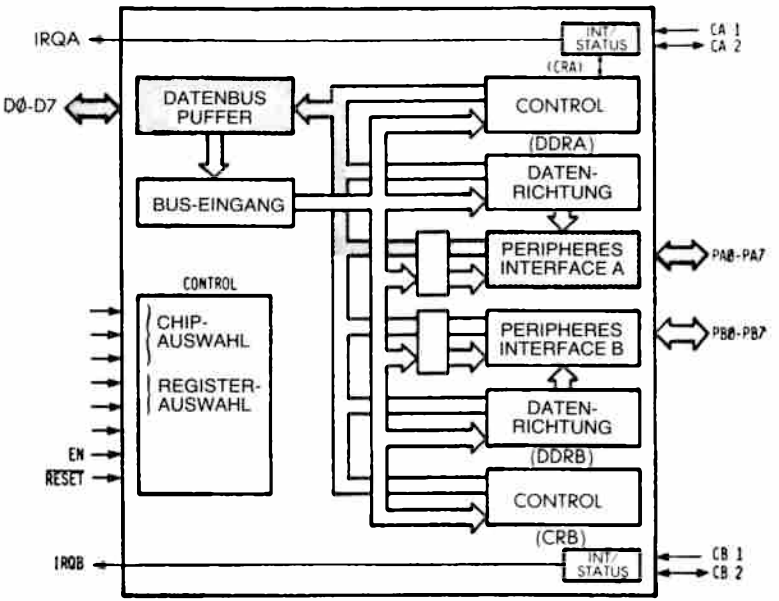


Abb. 7.5: Der Gebrauch einer PIO – Lesen einer Eingabe

Programmierung einer PIO

Wenn man einen PIO-Kanal als Eingang verwenden will, dann ergibt sich typischerweise folgender Ablauf:

Laden des Steuer-Registers

Dies erledigt man durch einen programmierten Transfer zwischen einem Z80-Register (normalerweise dem Akkumulator) und dem Steuer-Register der PIO. Damit werden die Optionen und die Betriebsart gesetzt (siehe Abb. 7.2). Dies macht man normalerweise einmal am Anfang eines Programms.

Laden des Richtungsregisters

Dies legt die Richtung fest, in der die Ein-/Ausgabe-Leitungen benutzt werden (siehe Abb. 7.3).

Lesen des Status

Das Statusregister zeigt an, ob ein gültiges Byte zur Eingabe verfügbar ist (siehe Abb. 7.4).

Lesen des Ports

Das Byte wird in den Z80 gelesen (siehe Abb. 7.5).

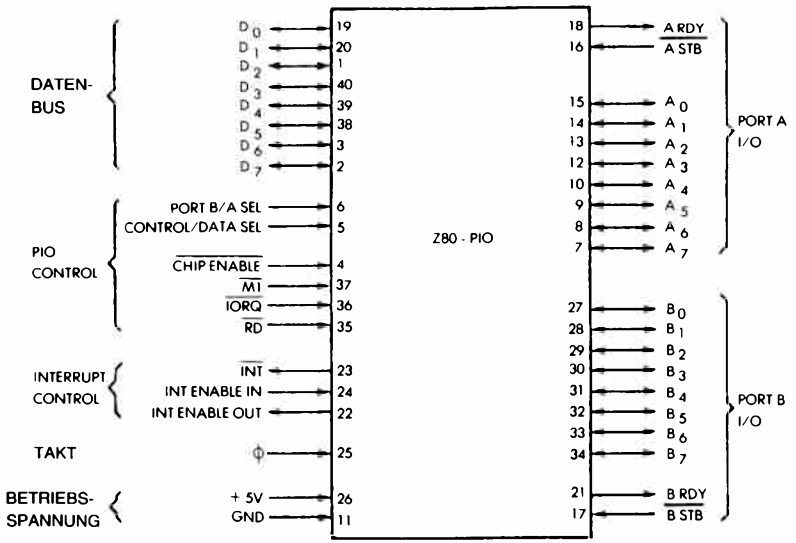


Abb. 7.6: Die Anschlußbelegung der Z80 PIO

Die Zilog Z80 PIO

Die Z80 PIO besitzt zwei Ports. Ihr Aufbau entspricht im wesentlichen dem Standardmodell, das wir eben beschrieben haben. Abb. 7.6 zeigt die tatsächliche Anschlußbelegung, in Abb. 7.7 ist ein Blockdiagramm dargestellt.

Jeder Port der PIO hat sechs Register: ein 8-Bit Eingaberegister, ein 9-Bit Ausgaberegister, ein 2-Bit Betriebsartenregister, ein 8-Bit Maskenregister, ein 8-Bit Ein-/Ausgabeauswahlregister (Richtungsregister) und ein 2-Bit Maskenkontrollregister. Die letzten drei Register werden nur benutzt, wenn die PIO im Bitmodus arbeitet.

Die PIO kann in einer von vier Betriebsarten arbeiten, die durch den Inhalt des Betriebsartenregisters (2 Bit) festgelegt wird. Dies sind: Byteausgabe, Byteeingabe, bidirektonaler Bytebus und Bitmodus.

Die beiden Bits im Maskenkontrollregister werden vom Programmierer geladen, und sie legen fest, ob der Zustand hoch oder niedrig des Peripheriegeräts überwacht werden soll, sowie die Bedingungen, unter denen ein Interrupt erzeugt wird.

Das 8-Bit Ein-/Ausgabe-Auswahlregister legt für jeden Anschluß fest, ob er Eingang oder Ausgang ist, wenn man im Bitmodus arbeitet.

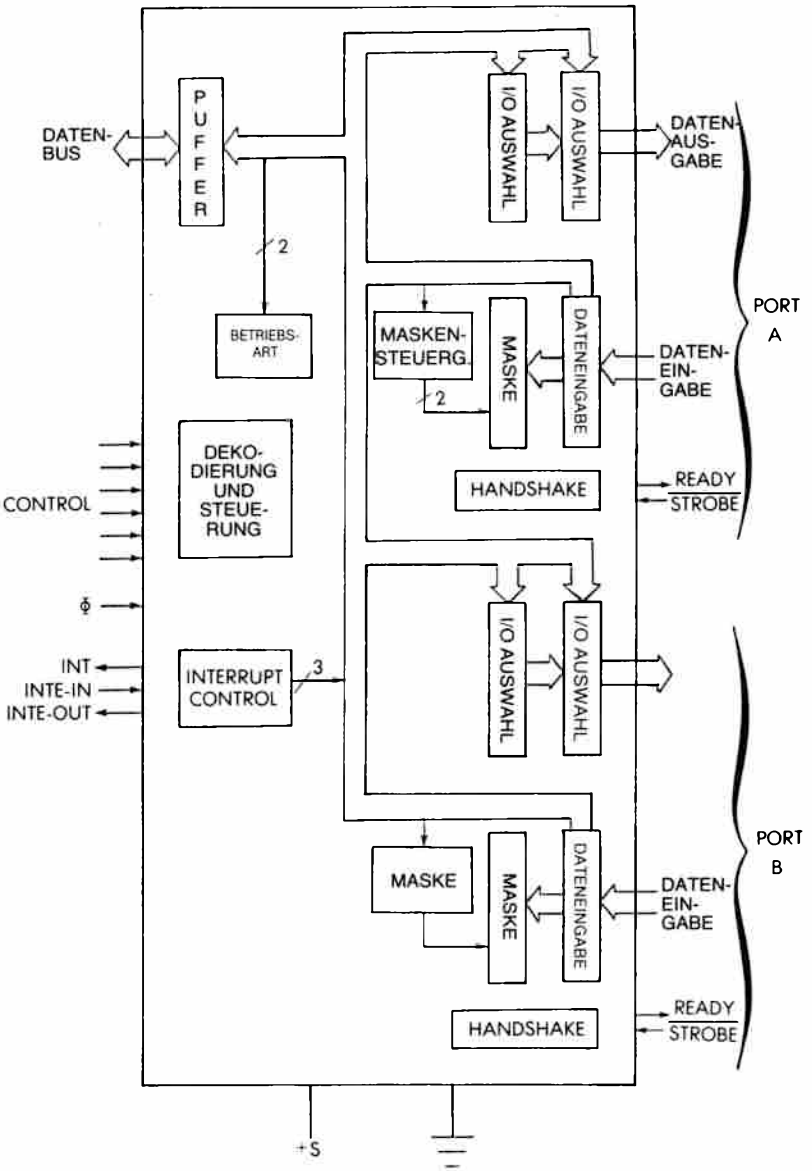


Abb. 7.7: Blockdiagramm der Z80 PIO

Programmierung der Zilog PIO

Eine typische Folge zur Programmierung einer PIO z. B. im Bit-Modus wäre folgendes:

Lade das Betriebsartenkontrollregister, um den Bit-Modus festzulegen.

Lade das Ein-/Ausgang-Auswahlregister von Port A um festzulegen, daß die Leitungen 0–5 Eingänge, die Leitungen 6 und 7 Ausgänge sind.

Danach wird ein Wort gelesen, indem der Inhalt des Eingangspuffers gelesen wird.

Zusätzlich könnte man das Maskenregister verwenden, um die Statusbedingungen festzulegen.

Für eine weitergehende Beschreibung der Arbeitsweise der PIO sei der Leser auf das begleitende Buch aus dieser Serie, das *Z80 Applikationsbuch* verwiesen.

Die Z80 SIO

Die SIO (serielle Ein-/Ausgabe) ist ein Peripheriebaustein mit zwei Kanälen, der entworfen wurde, um die asynchrone Kommunikation in serieller Form zu erleichtern. Sie enthält ein UART, d. h. einen universellen asynchronen Empfänger-Sender. Seine hauptsächliche Funktion ist die Wandlung seriell nach parallel und parallel nach seriell. Dieser Baustein ist jedoch mit hochentwickelten Eigenschaften ausgerüstet, wie automatische Behandlung komplexer byteorientierter Protokolle, wie IBM BSC (Binary Synchroneous Communication) oder auch HDLC (High Data Link Communication) und SDLC (Synchroneous Data Link Communication).
Zusätzlich kann sie in synchroner Betriebsart arbeiten, wie ein USRT, und CRC-Kodes erzeugen und testen. Sie bietet mehrere Betriebsarten zum Aufruf, für Interrupts und zur Blockübertragung. Die vollständige Beschreibung dieses Bausteins würde den Rahmen dieses einführenden Buchs bei weitem sprengen.

Andere Ein-/Ausgabebausteine

Da der Z80 üblicherweise den 8080 ersetzt, wurde er so konstruiert, daß er mit den meisten der gebräuchlichen 8080-Ein-/Ausgabebausteine arbeiten kann, wie auch mit den speziellen Ein-/Ausgabebausteinen, die Zilog herstellt. Alle Ein-/Ausgabebausteine des 8080 kann man zur Anwendung in Z80-Systemen in Betracht ziehen.

Zusammenfassung

Um die Ein-/Ausgabebausteine effektiv zu verwenden, ist es notwendig, daß man die Funktion jedes einzelnen Bits oder jeder Gruppe von Bits innerhalb der verschiedenen Steuer- Register versteht. Diese komplexen neuen Bausteine automatisieren eine Anzahl von Verfahren, die vorher softwaremäßig oder mit spezieller Logik ausgeführt werden mußten. Speziell ist in Bausteinen wie der SIO ein guter Teil der Handshaking-Funktionen automatisch eingebaut. Auch die Interrupterkennung und -behandlung kann intern realisiert sein. Mit der Information, die in diesem Kapitel geliefert wurde, sollte der Leser in der Lage sein, zu verstehen, welche Funktionen die Signale und Register haben. Natürlich werden noch weitere Bausteine eingeführt werden, die noch kompliziertere Algorithmen in der Hardware eingebaut haben.

8

Beispiele für Anwendungen

Einführung

Dieses Kapitel hat die Aufgabe, Ihre neuen Programmierkenntnisse zu testen, indem es eine Sammlung von nützlichen Programmen vorstellt. Diese Programme oder „Routinen" trifft man in Anwendungen häufig an und nennt sie deshalb „Hilfsprogramme". Sie verlangen eine Zusammenfassung der Kenntnisse und Techniken, die bisher vorgestellt wurden.

Wir werden Zeichen von einem Ein-/Ausgabegerät einlesen und sie auf verschiedene Arten verarbeiten. Zuerst wollen wir aber einen Speicherbereich löschen (dies mag unnötig sein – jedoch wird jedes dieser Programme nur als Programmierbeispiel vorgestellt).

Löschen eines Speicherbereichs

Wir wollen den Inhalt eines Speicherbereichs löschen (auf Null setzen), der bei der Adresse BASIS beginnt und bei der Adresse (BASIS + LAENGE-1) endet, wobei LAENGE kleiner als 256 sein soll.

Das Programm ist:

```
ZEROM  LD    B,LAENGE    Lade B mit LAENGE
       LD    A,0         Lösche A
       LD    HL,BASIS    Zeiger auf BASIS
CLEAR  LD    (HL),A      Lösche eine Stelle
       INC   HL          Zeiger auf nächste Stelle
       DEC   B           Dekrementiere Zähler
       JR    NZ,CLEAR    Ende des Bereichs?
       RET
```

Im obigen Beispiel ist angenommen, daß die Länge des Speicherbereichs gleich LANEGE ist. Das Registerpaar HL dient als Zeiger auf das laufende Wort, das gelöscht wird. Das Register B wird wie üblich als Zähler verwendet.

Der Akkumulator wird nur einmal mit dem Wert 0 geladen und dann in aufeinanderfolgende Speicherstellen kopiert.

In einem Speichertestprogramm könnte dieses Anwendungsprogramm beispielsweise dazu verwendet werden, den Inhalt eines Blocks auf Null zu setzen. Das Speichertestprogramm könnte danach testen, ob der Inhalt überall Null geblieben ist.

Das obige Beispiel war die direkte Realisierung eines Löschprogramms. Wir wollen es verbessern.

Das verbesserte Programm erscheint unten:

```
ZEROM LD    B,LAENGE
      LD    HL,BASIS
LOOP  LD    (HL),0
      INC   HL
      DJNZ  LOOP
      RET
```

Die beiden Verbesserungen wurden eingebaut, indem der Befehl LD A,0 entfernt und eine „Null" direkt in die Speicherstelle geladen wird, auf die H und L zeigen, und außerdem durch Verwendung des Z80-Spezialbefehls DJNZ.

Dieses Beispiel einer Verbesserung sollte zeigen, *daß jedes Programm, das geschrieben wurde, auch wenn es in Ordnung ist, in der Regel immer verbessert werden kann, wenn man es sorgfältig untersucht.* Die Vertrautheit mit dem vollständigen Befehlssatz ist wichtig, wenn man solche Verbesserungen einbringen will. Diese Verbesserungen sind nicht nur kosmetische Operationen. Sie erhöhen die Ausführungsgeschwindigkeit eines Programms, beanspruchen weniger Befehle und damit weniger Speicherplatz und erhöhen im allgemeinen auch die Lesbarkeit eines Programms und damit die Chancen, daß es korrekt ist.

Aufgabe 8.1: Schreiben Sie ein Speichertestprogramm, das einen Block von 256 Worten Null setzt, und dann überprüft, ob jede Stelle Null ist. Danach sollen lauter Einsen geschrieben und der Inhalt des Blocks überprüft werden. Danach wird 01010101 geschrieben und überprüft und letztlich 10101010.

Aufgabe 8.2: Verändern Sie das obige Programm so, daß es den Speicherbereich abwechselnd mit Nullen und Einsen füllt.

Wir wollen jetzt unsere Ein-/Ausgabebausteine abfragen um herauszufinden, welcher behandelt werden muß.

Abfrage von Ein-/Ausgabegeräten

Wir wollen annehmen, daß diese Ein-/Ausgabegeräte an unser System angeschlossen sind. Ihre Statusregister liegen bei den Adressen STATUS1, STATUS2 und STATUS3. Das Programm ist:

```
TEST    IN    A,(STATUS1)   Lies I/O-Status 1
        BIT   7,A           Teste das Bit „bereit" (Bit 7)
        JP    NZ,FOUND1     Springe zum Behandlungs
                            programm 1
        IN    A,(STATUS2)   dto. für Gerät 2
        BIT   7,A
        JP    NZ,FOUND2
        IN    A,(STATUS3)   dto. für Gerät 3
        BIT   7,A
        JP    NZ,FOUND3
        (Fehler-Ausgang)
```

Als Ergebnis des Befehls BIT wird das Bit Z des Flagregisters auf 1 gesetzt, wenn der Status nicht Null ist. Dann führt der Befehl JP NZ (springe, wenn nicht gleich Null) zu einer Verzweigung zur entsprechenden Routine FOUND.

Zeichen einlesen

Wir wollen annehmen, wir hätten gerade herausgefunden, daß von der Tastatur ein Zeichen anliegt. Wir wollen Zeichen in einem Speicherbereich namens BUFFER sammeln, bis wir ein Spezialzeichen genannt SPC erhalten, dessen Kode schon vorher definiert sein soll.

Das Unterprogramm GETCHAR holt ein Zeichen von der Tastatur (siehe Kapitel 6 für weitere Details) und legt es in den Akkumulator. Wir wollen annehmen, daß maximal 256 Zeichen geholt werden, bis ein SPC-Zeichen gefunden wird.

```
STRING  LD    HL,BUFFER     Zeiger auf BUFFER
NEXT    CALL  GETCHAR       Hole ein Zeichen
        CP    SPC           Teste auf ein Spezialzeichen
        JR    Z,OUT         Gefunden?
        LD    (HL),A        Speichere Zeichen in BUFFER
        INC   HL            Nächster Platz im BUFFER
        JR    NEXT          Hole nächstes Zeichen
OUT     RET
```

Aufgabe 8.3: Wir wollen dieses grundlegende Programm verbessern:
a – Gib das Zeichen wieder an das Gerät aus (z. B. an einen Fernschreiber).
b – Teste, daß die eingegebene Zeichenkette nicht länger als 256 Zeichen ist.

Wir haben jetzt eine Zeichenkette im Speicherpuffer. Wir wollen sie auf verschiedene Arten bearbeiten.

Test eines Zeichens

Wir wollen feststellen, ob das Zeichen in der Speicherstelle LOC gleich 0, 1 oder 2 ist:

```
ZOT     LD    A,(LOC)     Hole Zeichen
        CP    00H         Ist es Null?
        JP    Z,NULL      Sprung zur Routine
        CP    01H         Eine Eins?
        JP    Z,EINS
        CP    02H         Eine Zwei?
        JP    Z,ZWEI
        JP    NOTFND      Fehler
```

Wir lesen einfach das Zeichen und verwenden dann den Befehl CP, um seinen Wert zu testen.

Jetzt wollen wir einen anderen Test ausführen.

Test auf einen Zeichenbereich

Wir wollen feststellen, ob das ASCII-Zeichen in der Speicherstelle LOC eine Ziffer zwischen 0 und 9 ist:

```
BRACK   LD    A,(LOC)     Hole Zeichen
        AND   7FH         Blende Paritätsbit aus
        CP    30H         ASCII 0
        JR    C,OUT       Zeichen zu klein?
        CP    39H         ASCII 9
        JR    NC,OUT      Zeichen zu groß?
        CP    A           Setze Nullflag
OUT     RET
```

ASCII „0" wird hexadezimal durch „30" oder durch „D0" dargestellt, abhängig davon, ob das Paritätsbit verwendet wird oder nicht. Entsprechend wird ASCII „9" hexadezimal durch „39" oder „D9" dargestellt.

Der Zweck des zweiten Befehls in dem Programm ist es, das Bit 7, das Paritätsbit, zu löschen, für den Fall, daß es verwendet war, so daß das Programm in beiden Fällen anwendbar ist. Nach einem Vergleichsbefehl ist das Flag Z gesetzt, wenn sich Gleichheit ergab. Das Carrybit ist gesetzt, wenn ein Unterlauf auftrat, sonst zurückgesetzt. Mit anderen Worten, das Carrybit wird gesetzt, wenn der Wert der Konstanten, die in dem Befehl erscheint, größer ist als der Wert im Akkumulator. Ist er kleiner oder gleich, wird es zurückgesetzt („0").

Der letzte Befehl CP A mcht das Flag Z zu Null. Dieses Flag dient dazu, dem aufrufenden Programm anzuzeigen, ob das Zeichen in LOC in dem Intervall (0,9) liegt. Man kann natürlich auch andere Vereinbarungen benutzen, z. B. ein Zeichen in den Akkumulator laden, um das Ergebnis des Tests anzugeben.

Aufgabe 8.4: Ist das folgende Programm äquivalent zum obigen?

```
LD    A,(CHAR)
SUB   30H
JP    M,OUT
SUB   10H
JP    P,OUT
ADD   10H
```

Aufgabe 8.5: Stellen Sie fest, ob ein ASCII-Zeichen im Akkumulator ein Buchstabe des Alphabets ist.

Wenn Sie eine ASCII-Tabelle betrachten, dann werden Sie feststellen, daß die Parität oft verwendet wird. Beispielsweise ist der ASCII-Kode für „0" „0110000", ein 7-Bit-Kode. Verwenden wir jedoch zum Beispiel ungerade Parität, dann garantieren wir, daß die Gesamtzahl an Einsen in einem Wort ungerade ist. Der Kode wird dann zu „10110000". Links wurde eine zusätzliche „1" angefügt. Hexadezimal ist dies „B0". Wir wollen deshalb ein Programm entwerfen, daß die Parität erzeugt.

Erzeugen des Paritätsbits

Dises Programm erzeugt ein gerades Paritätsbit in der Stelle 7:

```
PARITY LD    A,(LOC)     Hole Zeichen
       AND   7FH         Lösche Paritätsbit
       JP    PE,OUT      Teste, ob Parität schon gerade
       OR    80H         Setze Paritätsbit
OUT    LD    (LOC),A     Speichere Ergebnis
```

Dieses Programm verwendet die interne Paritätsprüfung des Z80.

Der dritte Befehl: JP PE,OUT testet, ob die Parität des Wortes im Akkumulator schon gerade ist. Ist die Parität gerade „PE", dann wird dieser Sprung ausgeführt.

Ist die Parität nicht gerade, d. h. der Sprung wird nicht ausgeführt, dann ist die Parität ungerade und in der Stelle 7 muß eine „1" eingetragen werden. Dies ist der Zweck des vierten Befehls:

```
OR    80H
```

Schließlich wird der Wert, der sich ergab, in der Speicherstelle LOC abgelegt.

Codeumwandlung: ASCII nach BCD

Die Wandlung von ASCII nach BCD ist sehr einfach. Wir sehen, die hexadezimale Darstellung der ASCII-Zeichen 0 bis 9 ist 30 bis 39 bzw. B0 bis B9, abhängig von der Parität. Die BCD-Darstellung erhält man ein-

fach, wenn man die „3" oder das „B" wegläßt, d. h. das linke Nibble (4 Bit) maskiert:

```
ASCBCD CALL  BRACK          Teste, daß Zeichen zwischen
                            0 und 9
       JP    NZ,ILLEGAL     Ende, wenn unzulässiges Zeichen
       LD    A,(LOC)        Hole Zeichen
       AND   0FH            Maskiere oberes Nibble
       LD    (BCDCHAR),A    Speichere Ergebnis
```

Aufgabe 8.6: Schreiben Sie ein Programm zur Umwandlung von BCD nach ASCII.

Aufgabe 8.7: Schreiben Sie ein Programm zur Umwandlung von BCD nach Dual (schwieriger). Hinweis: $N_3\ N_2\ N_1\ N_0$ in BCD ist Dual $(((N_3 \times 10) + N_2) \times 10) + N_1) \times 10 + N_0$.

Verwenden Sie zur Multiplikation mit 10 ein Links Schieben (=x2), ein weiteres Links Schieben (=x4), dann ein ADC (=x5) und noch ein Links Schieben (=x10).

In der vollständigen BCD-Darstellung kann das erste Wort die Zahl der BCD-Ziffern enthalten, das nächste Nibble das Vorzeichen und jedes weitere Nibble eine BCD-Ziffer (wir nehmen an, daß kein Dezimalkomma vorhanden ist). Das letzte Nibble des Blocks kann unbelegt sein.

Umwandlung Hexadezimal nach ASCII

„A" enthalte eine hexadezimale Ziffer. Wir müssen einfach nur eine „3" (oder ein „B") im linken Nibble hinzufügen:

```
AND   OFH          Setze linkes Nibble Null
ADD   A,30H        ASCII
CP    39H          Korrektur notwendig?
JP    M,OUT
ADD   A,7          Korrektur für A bis F
```

Aufgabe 8.8: Wandle Hexadezimal nach ASCII um, wobei ein gepacktes Format vorliegen soll (zwei Hexadezimalziffern in A).

Das größte Element in einer Tabelle suchen

Die Anfangsadresse der Tabelle steht in der Speicherzelle BASE in Seite Null. Die erste Eintragung in der Tabelle ist die Zahl der Bytes, die sie enthält. Dieses Programm sucht das größte Element in der Tabelle. Sein Wert bleibt in A und seine Adresse wird in der Speicherzelle INDEX abgelegt.

Dieses Programm benutzt die Register A, B, H und L und verwendet indirekte Adressierung, so daß es eine Tabelle an beliebiger Stelle im Speicher durchsuchen kann (siehe Abb. 8.1).

```
MAX       LD    HL,BASE        Adresse der Tabelle
          LD    B,(HL)         Zahl der Bytes in der Tabelle
          LD    A,0            Lösche Maximalwert
          LD    (INDEX),HL     Initialisiere Index
          INC   HL             Nächste Eintragung
LOOP      CP    (HL)           Vergleiche Eintragung
          JR    NC,NOSWITCH    Sprung, falls kleiner als MAX
          LD    A,(HL)         Lade neuen Maximalwert
          LD    (INDEX),HL
NOSWITCH  INC   HL             Zeiger auf nächste Eintragung
          DEC   B              Dekrementiere Zähler
          JR    NZ,LOOP        Wiederhole, solange nicht Null
          RET
```

Dieses Programm testet zuerst die n-te Eintragung. Ist sie größer als Null, kommt die Eintragung nach A und ihre Adresse nach INDEX. Dann wird die (n-1)-te Eintragung getestet usw.

Dieses Programm funktioniert mit positiven ganzen Zahlen.

Aufgabe 8.9: Verändern Sie das Programm so, daß es auch mit negativen Zahlen in Zweierkomplement-Darstellung arbeitet.

Aufgabe 8.10: Funktioniert das Programm auch mit ASCII-Zeichen?

Aufgabe 8.11: Schreiben Sie ein Programm, das n Zahlen in aufsteigende Reihenfolge sortiert.

Aufgabe 8.12: Schreiben Sie ein Programm, das n Namen (aus je drei Zeichen) in alphabetischer Reihenfolge sortiert.

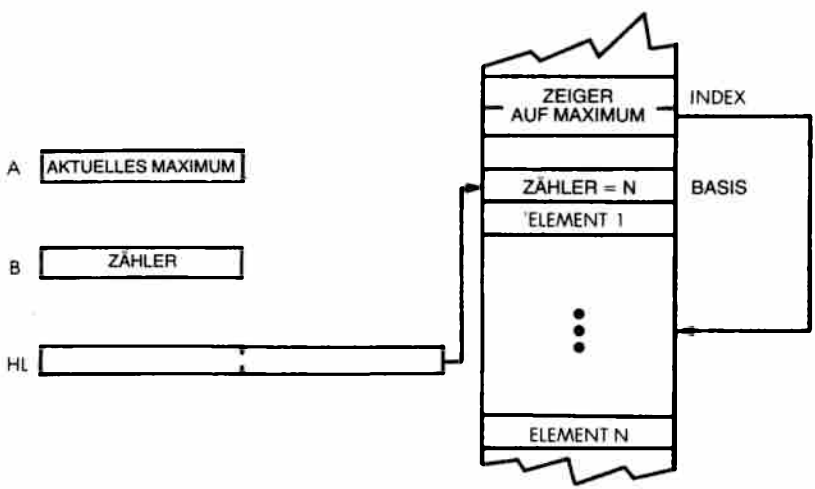


Abb. 8.1: Größtes Element in der Tabelle

Summe von N Elementen

Dieses Programm berechnet die 16-Bit-Summe von N positiven 8 Bit-Zahlen in einer Tabelle. Die Tabelle beginnt bei der Adresse BASE in der Seite Null. Die erste Eintragung in der Tabelle enthält die Anzahl der Elemente. Die 16-Bit-Summe wird in den Speicherzellen SUMLO und SUMHI abgelegt. Sollte die Summe länger als 16 Bit sein, erhält man nur die unteren 16 Bit. (Man sagt, die oberen Bits werden abgeschnitten.)

Das Programm verändert die Register A, B, H, L und IX. Es setzt ein Maximum von 256 Elementen voraus (siehe Abb. 8.2).

```
SUMIG    LD    HL,BASE        Zeiger auf Tabellenanfang
         LD    B,(HL)         Lies Länge in den Zähler
         INC   HL             Zeiger auf erste Eintragung
         LD    IX,SUMLO       Zeiger auf Ergebnis, unten
         LD    A,0            Lösche Ergebnis
         LD    (IX+0),A       unten
         LD    (IX+1),A       und oben
ADLOOP   LD    A,(HL)         Hole Eintragung
         ADD   A,(IX+0)       Berechne Teilergebnis
         LD    (IX+0),A       Speichere es
         JR    NC,NOCARRY     Test auf Übertrag
         INC   (IX+1)         Addiere carry ins obere Byte
NOCARRY  INC   HL             Zeiger auf nächste Eintragung
         DEC   B              Dekrementiere Bytezähler
         JR    NZ,ADLOOP      Weiter addieren, bis fertig
         RET
```

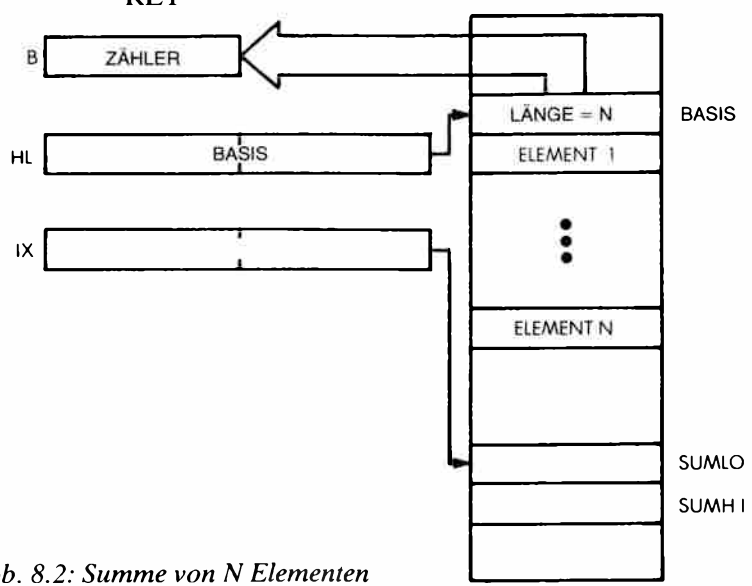


Abb. 8.2: Summe von N Elementen

Dieses Programm ist einfach und sollte sich selbst erklären.

Aufgabe 8.13: Verändern Sie dieses Programm, um
a – eine 24-Bit-Summe zu berechnen
b – eine 32-Bit-Summe zu berechnen
c – einen Überlauf anzuzeigen.

Berechnung einer Prüfsumme

Eine Prüfsumme ist eine Ziffer oder ein Satz von Ziffern, die aus einem Satz aufeinanderfolgender Zeichen berechnet werden. Die Prüfsumme wird zu dem Zeitpunkt berechnet, zu dem die Daten gespeichert werden, und am Ende abgelegt. Um zu überprüfen.daß die Daten unversehrt sind, liest man die Daten, berechnet die Prüfsumme neu und vergleicht sie mit dem gespeicherten Wert. Ein Unterschied zeigt einen Fehler an.

Dazu werden verschiedene Algorithmen verwendet. Hier wollen wir alle Bytes in einer Tabelle von N Elementen exklusiv-OR verknüpfen und das Ergebnis im Akkumulator stehen lassen. Wie üblich beginnt die Tabelle bei der Adresse BASE. Die erste Eintragung in die Tabelle ist die Anzahl N der Elemente. Das Programm verändert A, B, H und L. N muß kleiner als 256 sein.

```
CHECKSUM LD    HL,BASE     Lies Tabellenanfang nach HL
         LD    B,(HL)      Hole N = Länge
         XOR   A           Lösche Prüfsumme
         INC   HL          Zeiger auf das erste Element
CHLOOP   XOR   (HL)        Berechne Prüfsumme
         INC   HL          Zeiger auf nächstes Element
         DEC   B           Dekrementiere Zähler
         JR    NZ,CHLOOP   Wiederhole, falls nicht fertig
         LD    (HL),A      Speichere Prüfsumme am
                           Tabellenende
         RET
```

Zählen der Nullen

Dieses Programm zählt die Anzahl der Nullen in unserer üblichen Tabelle und speichert sie bei der Adresse TOTAL. Es verändert A, B, C, H und L.

```
ZEROS    LD    HL,BASE     Zeiger auf Tabelle
         LD    B,(HL)      Lies Länge in den Zähler
         LD    C,0         Summe der Nullen
         INC   HL          Zeiger auf erste Eintragung
ZLOOP    LD    A,(HL)      Hole Element
         OR    0           Setze Nullflag
         JR    NZ,NOTZ     Ist es Null?
         INC   C           Wenn ja, inkrementiere
                           Nullenzähler
```

```
NOTZ    INC   HL          Zeiger auf nächste Eintragung
        DEC   B           Dekrementiere Längenzähler
        JR    NZ,ZLOOP
        LD    A,C
        LD    (TOTAL),A   Speichere Ergebnis
```

Aufgabe 8.14: Ändern Sie dieses Programm zum Zählen
a – der Anzahl der Sterne (Zeichen „*“)
b – der Anzahl der Buchstaben des Alphabets
c – der Anzahl der Ziffern zwischen „0“ und „9“.

Übertragung von Blöcken

Wir wollen jede dritte Eintragung aus dem Quellblock bei der Adresse FROM herausgreifen und in einem Block bei der Adresse TO ablegen:

```
FER3    LD    HL,FROM
        LD    DE,TO       Setze Zeiger
        LD    BC,SIZE
LOOP    LDI               Automatischer Transfer
        INC   HL
        INC   HL          Überspringe zwei Eintragungen
        JR    PE,LOOP
```

Transfer eines Blocks von BCD-Zahlen

Wir wollen BCD-Ziffern, d. h. 4-Bit-Nibbles in den Speicher hineinschieben (siehe Abb. 8.3). Das Programm erscheint unten:

```
DMOV    LD    B,COUNT
        LD    HL,BLOCK
        XOR   A           A=0
LOOP    RLD
        INC   HL          Zeiger auf nächstes Byte
        DJNZ  LOOP        Dekrementiere Schleifenzähler
                          bis Null
```

Das Programm verwendet den Befehl RLD, den wir bisher noch nicht benutzt haben. RLD rotiert eine BCD-Ziffer zwischen A und (HL). (HL) oder M bezeichnet den Inhalt der Speicherzellen, auf die H und L zeigen.

M unten kommt nach M oben
M oben kommt nach A unten
A unten kommt nach M unten

Hier beziehen sich „unten“ und „oben“ auf ein 4-Bit-Nibble.

Um den leistungsfähigen Befehl DJNZ verwenden zu können, dient das Register B als Ziffernzähler. HL wird auf den Blockanfang gesetzt.

A wird dazu verwendet, die linke Ziffer zu speichern, die bei jeder Rotation zwischen zwei Zugriffen auf den Block versetzt wird.

Vereinbarungsgemäß wird am oberen Ende des Blockes „O“ eingetragen.

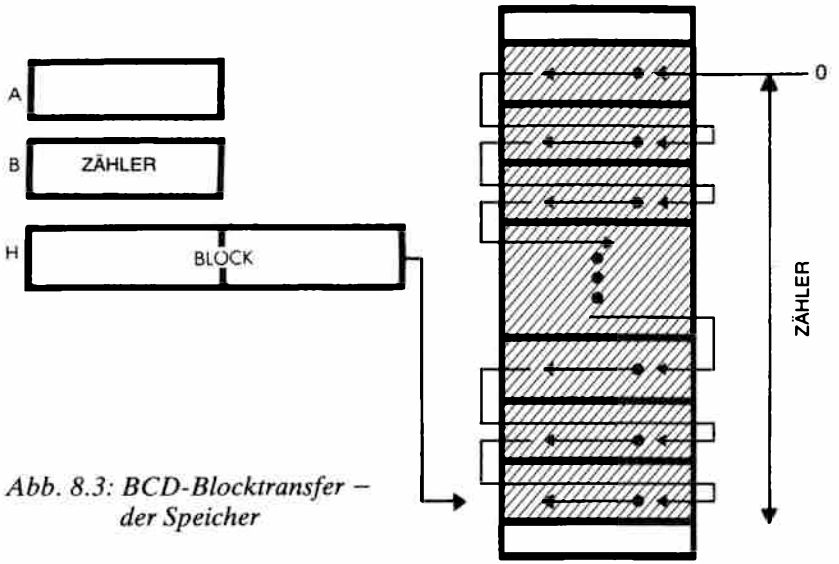


Abb. 8.3: BCD-Blocktransfer – der Speicher

Vergleich zweier Vorzeichenbehafteter 16 Bit-Zahlen

IX zeigt auf die erste Zahl N1.
IY zeigt auf N2 (siehe Abb. 8.4)

Das Programm setzt das Übertragsflag, wenn N1<N2, und das Nullflag, wenn N1=N2.

```
COMP    LD    B,(IX+1)     Hole Vorzeichen von N1
        LD    A,B
        AND   80H          Teste Vorzeichen, lösche Carry
        JR    NZ,NEGM1     N1 ist negativ
        BIT   7,(IY+1)
        RET   NZ           N2 ist negativ
        LD    A,B
        CP    (IY+1)       Beide Vorzeichen positiv
        RET   NZ
        LD    A,(IX)
        CP    (IY)
        RET
NEGM1   XOR   (IY+1)
        RLA                Vorzeichenbit ins Carry
        RET   C            Vorzeichen verschieden
        LD    A,B
        CP    (IY+1)       Beide Vorzeichen negativ
        RET   NZ
        LD    A,(IX)
        CP    (IY)
        RET
```

Das Programm testet zuerst die Vorzeichen von N1 und N2. Ist N1 negativ, wird zu NEGM1 gesprungen. Sonst wird der obere Teil des Programms ausgeführt.

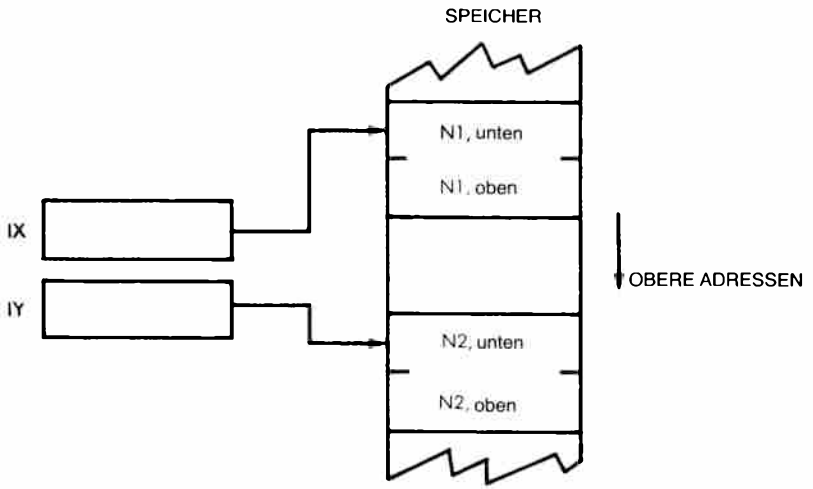


Abb. 8.4: Vergleich zweier vorzeichenbehafteter Zahlen

Beachten Sie, daß der Befehl BIT in der fünften Zeile dazu verwendet wird, das Vorzeichenbit von N2 direkt im Speicher zu testen:

```
        BIT   7,(IY+1)
```

Für N1 hätte man ebenso vorgehen können, aber wir werden den Wert von N1 in Kürze brauchen. Es ist deshalb einfacher, N1 aus dem Speicher zu lesen und in B abzulegen:

```
COMP    LD    B,(IX+1)
```

Es ist nötig, N1 in B zu speichern, da AND den Inhalt von A zerstört:

```
        LD    A,B
        AND   80H
```

Beachten Sie auch, daß ein bedingter Rücksprung verwendet wird (Zeile 6):

```
        RET   NZ
```

Dies ist eine leistungsfähige Eigenschaft des Z80, die die Programmierung vereinfacht.

Beachten Sie, daß der Vergleichsbefehl in indizierter Adressierung direkt im Speicher ausgeführt wird:

```
        CP    (IY+1)
```

Wenn man die beiden Zahlen vergleicht, wird zuerst das am meisten signifikante Byte verglichen, danach das weniger signifikante.

Beachten Sie die ausgiebige Anwendung der indizierten Adressierung in diesem Programm, die zu effizientem Kode führt.

Bubble-Sort

Bubble-Sort („Blasensortieren") ist ein Sortierverfahren, das man verwendet, um Elemente einer Tabelle in steigender oder fallender Ordnung zu sortieren. Das Bubble-Sort-Verfahren leitet seinen Namen davon ab, daß das kleinste Element wie eine Luftblase zum oberen Ende der Tabelle steigt. Immer wenn es mit einem „schwereren" Element zusammenstößt, überspringt es dieses.

Ein praktisches Beispiel für Bubble-Sort ist in Abb. 8.5 gezeigt. Die Liste, die sortiert werden soll, enthält: (10, 5, 0, 2, 100), und sie soll in aufsteigender Reihenfolge sortiert werden („0" oben). Der Algorithmus ist einfach, das Flußdiagramm ist in Abb. 8.7 dargestellt.

Die beiden obersten (oder die beiden untersten) Elemente werden verglichen. Ist das untere kleiner („leichter") als das obere, werden sie vertauscht, sonst nicht. Zur praktischen Anwendung wird der Austausch, wenn er stattfindet, in seinem Flag namens „EXCHANGED" gespeichert. Das Verfahren wird dann mit dem nächsten Elementepaar wiederholt, usw. bis alle Elemente paarweise verglichen wurden.

Der erste Durchgang ist in den Schritten 1, 2, 3, 4, 5 und 6 in Abb. 8.5 dargestellt. Dabei wurde von unten nach oben vorgegangen. (Entsprechend hätten wir auch von oben nach unten vorgehen können.)

Wurden keine Elemente vertauscht, ist die Tabelle fertig sortiert. Trat ein Austausch auf, wird der Durchgang wiederholt.

Wenn wir Abb. 8.6 betrachten, dann sehen wir, daß in diesem Beispiel vier Durchgänge nötig sind.

Das Verfahren ist einfach und wird häufig angewendet.

Eine Schwierigkeit besteht noch in der Durchführung des Austauschs. Wenn wir A und B vertauschen wollen, dann können wir nicht schreiben

```
A = B
B = A,
```

da dann der alte Wert von A verloren wäre (probieren Sie es an einem Beispiel).

Die richtige Lösung ist es, den Wert von A in einem Register oder einer Speicherstelle zwischenzuspeichern:

```
TEMP = A
A    = B
B    = TEMP
```

Es funktioniert (probieren Sie es an einem Beispiel)! Man nennt dieses Vorgehen zyklische Vertauschung. Alle Programme führen eine Vertauschung nach diesem Verfahren durch. Diese Technik ist in dem Flußdiagramm in Abb. 8.7 veranschaulicht.

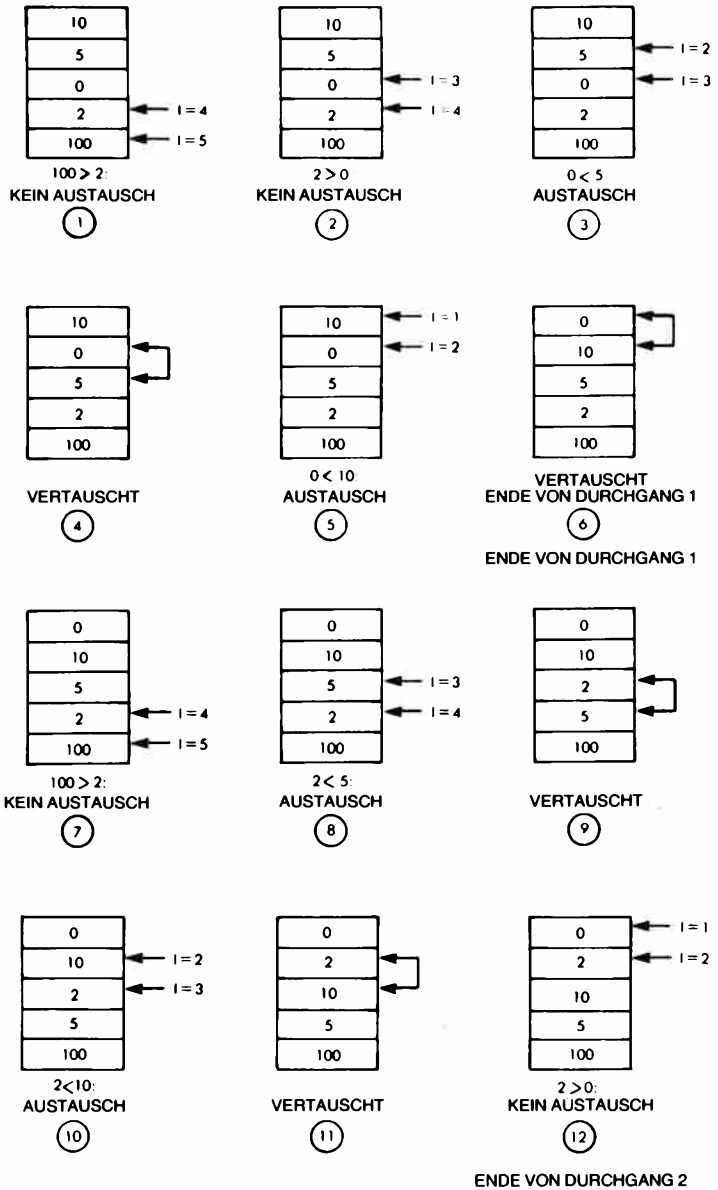


Abb. 8.5: Bubble-Sort Beispiel: Schritte 1 bis 12

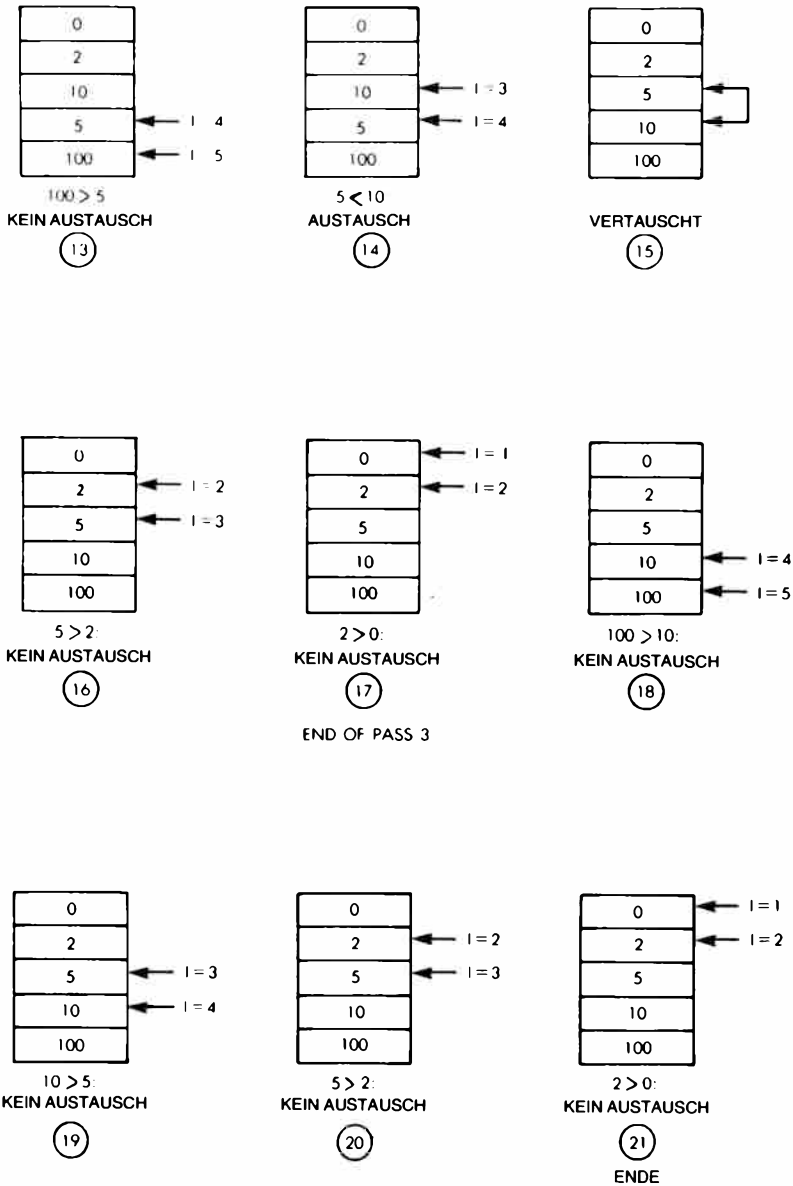


Abb. 8.6: Bubble-Sort Beispiel: Schritte 13 bis 21

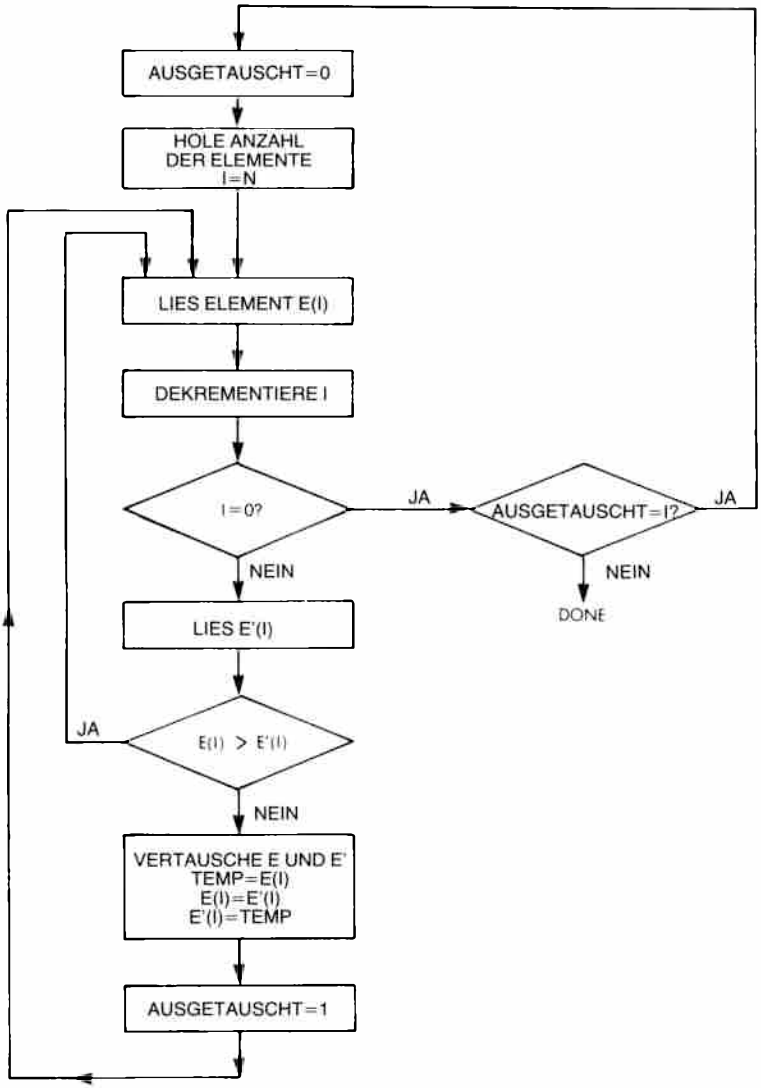


Abb. 8.7: Bubble-Sort Flußdiagramm

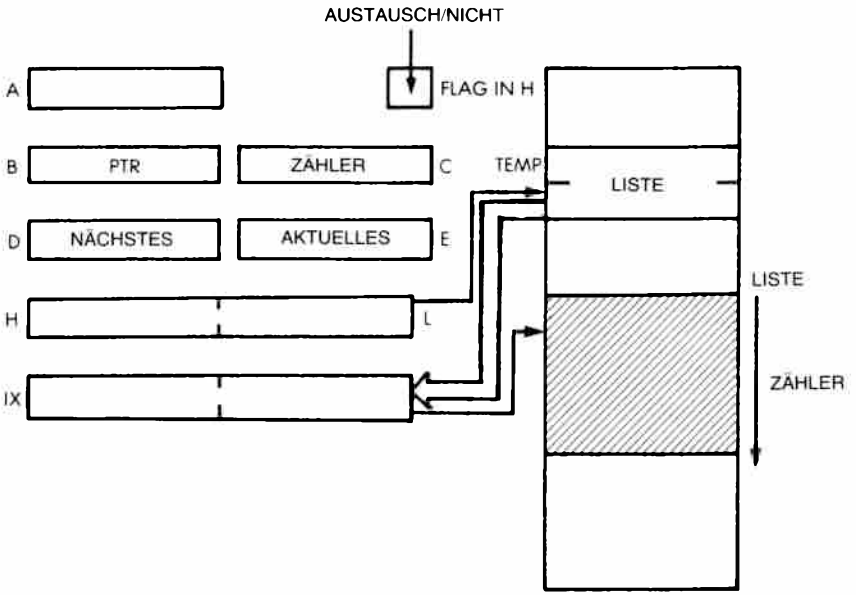


Abb. 8.8: Bubble-Sort

Abbildung 8.8 zeigt die Belegung der Register und des Speichers. Das Programm ist:

```
BUBBLE   LD    (TEMP),HL      TEMP = (HL)
         LD    IX,(TEMP)      IX=(HL)
         RES   FLAG,H         Austauschflag = 0
         LD    B,C
         DEC   B
NEXT     LD    A,(IX)
         LD    D,A            D = aktuelle Eintragung
         LD    E,(IX+1)       E = nächste Eintragung
         SUB   E              Vergleich
         JR    NC,NOSWITCH    Sprung, wenn aktuelle
                              > nächste Eintragung
XCHANGE  LD    (IX),E         Speichere nächste in
                              aktuelle Eintragung
         LD    (IX+1),D       Speichere aktuelle in
                              nächste Eintragung
         SET   FLAG,H         Austauschflag = 1
NOSWITCH INC   IX             Nächste Eintragung
         DJNZ  NEXT           Dekr. B, weiter bis Null
         BIT   FLAG,H         Vertauschung ausgeführt?
         JR    NZ,AGAIN       Wiederholung, falls FLAG = 1
         RET
```

Zusammenfassung

In diesem Kapitel wurden gebräuchliche Anwendungsprogramme vorgestellt, die Kombinationen von Techniken verwenden, die wir in den vorhergehenden Kapiteln beschrieben haben. Sie sollten Ihnen jetzt erlauben, mit dem Entwurf eigener Programme zu beginnen. Viele dieser Programme haben eine spezielle Datenstruktur benutzt, die Tabelle. Es gibt noch weitere Möglichkeiten zur Strukturierung von Daten, und diese werden im folgenden Kapitel vorgestellt.

9

Datenstrukturen

1. Teil – Theorie

Einführung

Der Entwurf eines guten Programms schließt zwei Aufgaben ein: den *Entwurf des Algorithmus* und den *Entwurf der Datenstrukturen*. Die meisten einfachen Programme enthalten keine wesentlichen Datenstrukturen, so daß es das Hauptziel ist, Algorithmen zu entwerfen und sie in einer gegebenen Maschinensprache effizient zu kodieren, wenn man Programmieren lernt. Das haben wir bisher getan. Der Entwurf komplexerer Programme setzt jedoch auch ein Verständnis von Datenstrukturen voraus. In diesem Buch wurden schon zwei Datenstrukturen benutzt: Tabelle und Stapel. Die Aufgabe dieses Kapitels ist es, weitere allgemeine Datenstrukturen vorzustellen, die Sie vielleicht verwenden können. Dieses Kapitel ist vollkommen uanbhängig von dem gewählten Mikroprozessor. Es ist theoretisch und enthält die logische Organisation von Daten in dem System. Es gibt spezielle Bücher über Datenstrukturen, wie es auch spezielle Bücher über effiziente Multiplikation, Division und andere nützliche Algorithmen gibt. Deshalb beschränkt sich dieses Kapitel nur auf das Wesentliche. Es erhebt nicht den Anspruch auf Vollständigkeit. Es sollen nun die gebräuchlichsten Datentrukturen vorgestellt werden.

Zeiger

Ein Zeiger ist eine Zahl, die dazu dient, die Adresse der tatsächlichen Daten anzugeben. Jeder Zeiger ist eine Adresse. Jedoch wird nicht jede Adresse Zeiger genannt. Eine Adresse ist nur dann ein Zeiger, wenn sie auf gewisse Typen von Daten oder von strukturierter Information zeigt. Wir haben schon einen speziellen Typ von Zeiger eingeführt, den Stapelzeiger, der auf das oberste Stapelelement zeigt (oder unmittelbar darüber). Wir werden sehen, daß der Stapel eine gebräuchliche Datenstruktur ist, genannt LIFO Struktur.

Ein weiteres Beispiel: Wenn wir indirekte Adressierung verwenden, ist die indirekte Adresse immer ein Zeiger auf die Daten, auf die man zugreifen will.

Aufgabe 9.1: Untersuchen Sie die Abbildung 9.1. In der Adresse 15 im Speicher steht ein Zeiger auf die Tabelle T. Die Tabelle T beginnt bei der Adresse 500. Was ist der tatsächliche Inhalt des Zeigers auf T?

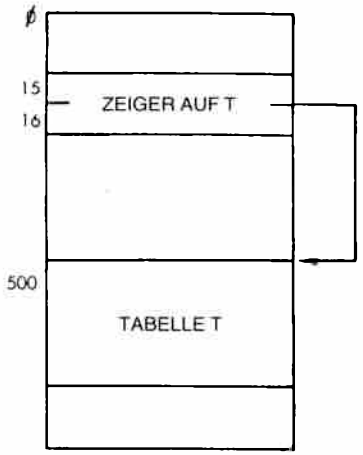


Abb. 9.1: Ein indirekter Zeiger

Listen

Fast alle Datenstrukturen sind in Form verschiedener Arten von Listen organisiert.

Sequentielle Listen

Eine sequentielle Liste, eine Tabelle oder ein Block ist die wahrscheinlich einfachste Datenstruktur, und wir haben sie auch schon verwendet. Tabellen sind normalerweise nach einem bestimmten Kriterium geordnet, z. B. alphabetisch oder numerisch. Dann kann man leicht ein Element in der Tabelle finden, wenn man z. B. indizierte Adressierung verwendet, wie wir es schon getan haben. Als Block bezeichnet man normalerweise eine Gruppe von Daten, die festgesetzte Begrenzungen hat, deren Inhalt aber nicht geordnet ist. Er kann eine Zeichenkette enthalten, er mag einen Sektor auf einer Platte bilden, oder er kann ein bestimmter logischer Bereich im Speicher (genannt Segment) sein. In solchen Fällen ist es nicht leicht, auf ein beliebiges Element des Blocks zuzugreifen.

Um den Zugriff auf Blöcke von Information zu erleichtern, benutzt man Inhaltsverzeichnisse.

Inhaltsverzeichnisse

Ein Inhaltsverzeichnis (englisch: directory) ist eine Liste von Tabellen oder Blöcken. Zum Beispiel ist das Dateisystem normalerweise nach einem Inhaltsverzeichnis strukturiert. Als einfaches Beispiel soll das Hauptinhaltsverzeichnis des Systems eine Liste mit den Namen der Benutzer enthalten. Dies ist in Abb. 9.2 veranschaulicht. Die Eintragung für den Benutzer „John" zeigt auf das Inhaltsverzeichnis von Johns Dateien. Dieses ist wieder eine Tabelle von Zeigern. In diesem Fall haben wir ein Inhaltsverzeichnis aus zwei Ebenen entworfen. Ein flexibles System von Inhaltsverzeichnissen läßt es zu, daß weitere darunterliegende Inhaltsverzeichnisse eingeschlossen werden, wenn es der Benutzer für günstig hält.

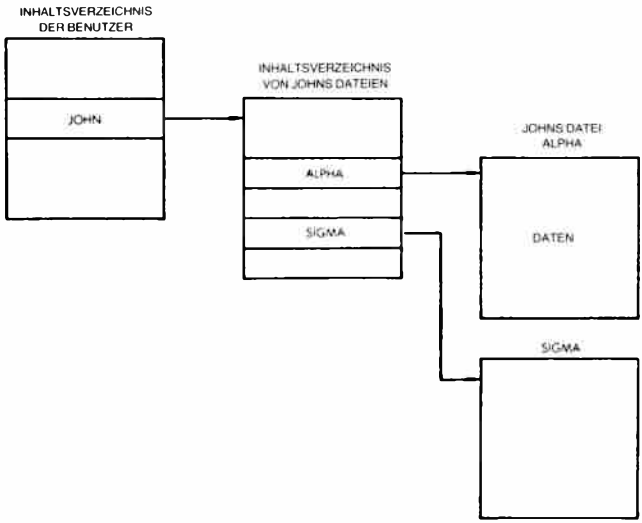


Abb. 9.2: Struktur eines Inhaltsverzeichnisses

Verkettete Liste

In einem System hat man oft Blöcke von Information, die Daten, Ereignisse oder andere Strukturen enthalten, die man nicht leicht verschieben kann. Könnte man dies tun, dann würden wir sie wahrscheinlich in einer Tabelle sammeln, um sie zu sortieren oder zu strukturieren. Das Problem ist aber, daß wir sie dort stehen lassen wollen, wo sie stehen, und trotzdem eine Ordnung einführen wollen, wie erstes, zweites, drittes, viertes Element. Dieses Problem kann man mit einer verketteten Liste lösen. Das Konzept der verketteten Liste zeigt Abbildung 9.3. In der Abbildung sehen wir, daß ein Listenzeiger, genannt ERSTERBLOCK, auf den Anfang des ersten Blocks zeigt. Eine festgesetzte Stelle im er-

sten Block, z. B. das erste oder das letzte Wort, enthält einen Zeiger auf Block 2, genannt PTR1. Dieses Verfahren wiederholt sich dann bei Block 2 und bei Block 3. Da Block 3 die letzte Eintragung in der Liste ist, enthält PTR3 vereinbarungsgemäß ein Nullzeichen oder er zeigt auf sich selbst, so daß man das Ende der Liste erkennen kann. Diese Struktur ist wirtschaftlich, da sie nur wenige Zeiger benötigt (einen pro Block), und den Benutzer von der Aufgabe befreit, die Blöcke im Speicher physikalisch verschieben zu müssen.

Abb. 9.3: Eine verkettete Liste

Wir wollen z. B. untersuchen, wie ein Block eingefügt wird. Abbildung 9.4 veranschaulicht dies. Wir wollen annehmen, daß der neue Block die Adresse NEUERBLOCK hat, und daß er zwischen Block 1 und Block 2 eingefügt werden soll. Der Zeiger PTR1 wird einfach auf den Wert NEUERBLOCK geändert, so daß er jetzt auf den Block X zeigt. PTRX wird mit dem früheren Wert von PTR1 geladen, d. h. er zeigt auf Block 2. Die anderen Zeiger in der Struktur bleiben unverändert. Wir sehen, daß zum Einfügen eines neuen Blocks nur zwei Zeiger in der Struktur geändert werden mußten. Dies ist natürlich effizient.

Aufgabe 9.2: Zeichnen Sie ein Diagramm, wie Block 2 aus dieser Struktur entfernt würde.

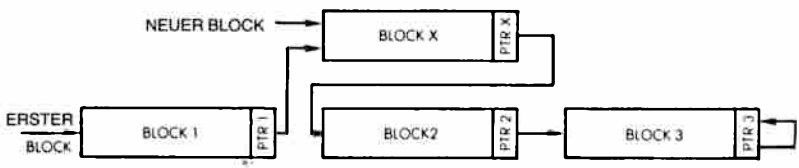


Abb. 9.4: Einfügen eines Blocks

Um spezielle Arten von Zugriff, Einfügen und Löschen zu erleichtern, wurden verschiedene Typen von Listen entwickelt. Wir wollen einige der gebräuchlichsten Typen verketteter Listen untersuchen.

Schlange

Eine Schlange wird formal FIFO oder First-In-First-Out genannt. Abbildung 9.5 veranschaulicht eine Schlange. Um das Diagramm zu erklären, können wir beispielsweise annehmen, daß der Block auf der linken

Seite ein Treiberprogramm für ein Ausgabegerät wie einen Drucker ist. Die Blöcke auf der rechten Seite sind Anforderungen von verschiedenen Programmen oder Routinen, Zeichen zu drucken. Die Reihenfolge, in der sie bearbeitet werden, ist die Ordnung in der Warteschlange. Man sieht, daß zuerst der Block 1, dann der Block 2 und danach Block 3 bearbeitet wird. Bei einer Schlange gilt die Vereinbarung, daß neue Anforderungen am Ende angefügt werden. Hier werden sie hinter Block 3 angehängt. Dies garantiert, daß der Block, der als erster in die Schlange eingefügt wird, als erster bearbeitet wird. Schlangen werden in Computern normalerweise dann verwendet, wenn verschiedene Vorgänge warten müssen, z. B. auf den Prozessor oder auf Ein-/Ausgabgeräte.

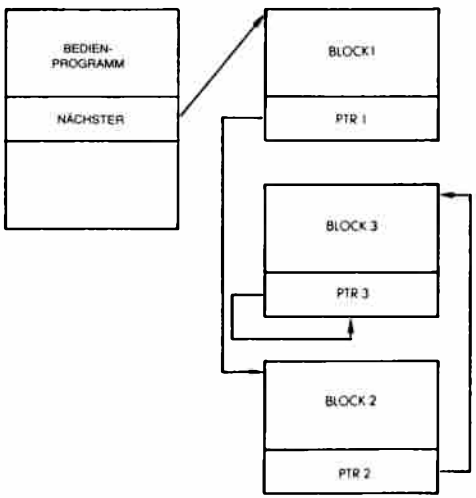


Abb. 9.5: Eine Schlange

Stapel

Die Stapelstruktur haben wir dieses Buch hindurch schon im einzelnen untersucht. Sie ist eine Last-In-First-Out (LIFO) Struktur. Das Element, das als letztes auf dem Stapel abgelegt wurde, wird als erstes wieder geholt. Ein Stapel kann entweder als sortierter Block oder als Liste aufgebaut werden. Da die meisten Stapel in Mikroprozessoren für Ereignisse mit hoher Geschwindigkeit verwendet werden, wie Unterprogramme und Interrupts, wird für den Stapel üblicherweise ein zusammenhängender Block verwendet und keine verkettete Liste.

Verkettete Liste oder Block

Entsprechend könnte man auch die Schlange als Block reservierter Speicherplätze realisieren. Der Vorteil eines zusammenhängenden Blocks

ist ein schneller Zugriff und der Wegfall der Zeiger. Der Nachteil ist, daß man üblicherweise einen ziemlich großen Block reservieren muß, um der Größe der Struktur im ungünstigsten Fall zu entsprechen. Außerdem ist es schwierig, Elemente in den Block einzufügen oder aus dem Block zu entfernen. Da Speicher traditionell knapp ist, verwendet man Blöcke nur für Strukturen fester Größe oder für Strukturen, auf die man mit maximaler Geschwindigkeit zugreifen muß, wie auf den Stapel.

Ringliste

Eine Ringliste ist eine verkettete Liste, in der die letzte Eintragung wieder auf den Anfang zeigt. Abbildung 9.6 veranschaulicht das. Bei einer Ringliste hat man oft einen Zeiger auf den aktuellen Block. Wenn Ereignisse oder Programme auf Bearbeitung warten, wird der Zeiger auf das aktuelle Ereignis um eine Stelle nach links oder rechts verschoben. Eine Ringliste entspricht üblicherweise einer Struktur, in der für alle Blöcke die gleiche Priorität angenommen wird. Eine Ringliste kann man aber auch anderen Strukturen untergeordnet verwenden, wenn man z. B. beim Suchen den Zugriff auf den ersten Block erleichtern will, nachdem man auf den letzten zugegriffen hatte.

Beispielsweise geht ein Programm beim Polling oft nach einer Ringliste vor. Es werden alle Peripheriegeräte der Reihe nach abgefragt und dann beginnt man wieder beim ersten.

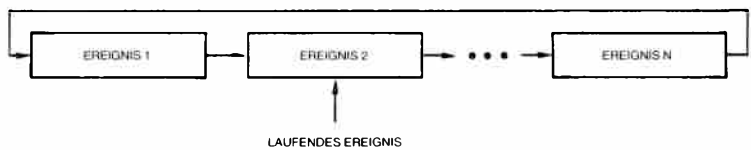


Abb. 9.6: Ringliste

Bäume

Wenn unter den Elementen einer Struktur ein logischer Zusammenhang besteht (dies nennt man üblicherweise eine Syntax), dann kann man eine Baumstruktur verwenden. Ein einfaches Beispiel dafür ist ein Stammbaum. Dies ist in Abb. 9.7 dargestellt. Man sieht, daß Herr Schmidt zwei Kinder hat, einen Sohn Robert und eine Tochter Jane. Jane wiederum hat drei Kinder: Liz, Tom und Phil. Tom hat zwei Kinder: Max und Chris. Robert dagegen, auf der linken Seite der Abbildung, hat keine Nachkommen.

Dies ist ein strukturierter Baum. In Abb. 9.2 haben wir tatsächlich schon ein Beispiel für einen einfachen Baum eingeführt. Die Struktur des Inhaltsverzeichnisses war ein Baum mit zwei Ebenen. Bäume verwendet man vorzugsweise dann, wenn Elemente in eine feste Struktur eingeordnet werden können. Dies erleichtert das Einfügen und Löschen. Zusätz-

lich kann man Information strukturiert so gruppieren, daß die spätere Bearbeitung, z. B. durch Compiler oder Interpreter erleichtert wird.

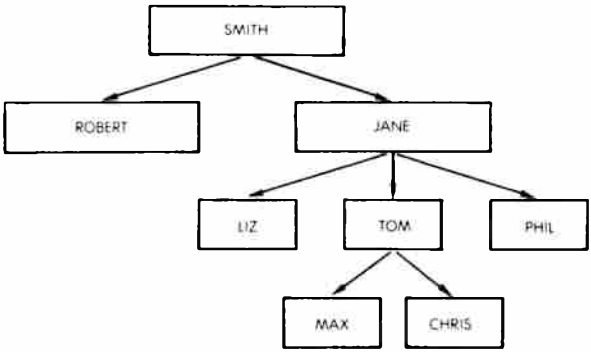


Abb. 9.7: Stammbaum

Doppelt verkettete Liste

Zwischen den Elementen einer Liste kann man zusätzliche Verkettungen einbauen. Das einfachste Beispiel ist eine doppelt verkettete Liste. Abbildung 9.8 veranschaulicht das. Wir sehen, daß wir von links nach rechts die übliche Folge von Verkettungen haben, außerdem eine zusätzliche Folge von Verkettungen von rechts nach links. Der Vorteil ist, daß man genauso einfach auf den Vorgänger des Elements zugreifen kann, das momentan bearbeitet wird, wie auf den Nachfolger. Dies kostet für jeden Block einen zusätzlichen Zeiger.

Abb. 9.8: Doppelt verkettete Liste

Suchen und Sortieren

Das Suchen und Sortieren von Elementen einer Liste hängt direkt von der Art der Struktur ab, die für die Liste verwendet wurde. Für die häufig verwendeten Datenstrukturen wurden viele Suchverfahren entwikkelt. Wir haben schon die indizierte Adressierung verwendet. Dies ist möglich, wenn die Elemente einer Tabelle nach einem bestimmten Kriterium geordnet sind. Auf solche Elemente kann man über ihre Nummer zugreifen.

Sequentielle Suche heißt, einen fortlaufenden Block linear zu durchsuchen. Dies ist natürlich nicht effizient, muß aber eventuell angewendet

werden, wenn man kein besseres Verfahren hat, weil die Elemente nicht geordnet sind.

Binäre oder logarithmische Suchverfahren versuchen ein Element in einer sortierten Liste dadurch zu finden, daß das Suchintervall bei jedem Schritt halbiert wird. Wollen wir z. B. eine alphabetische Liste durchsuchen, können wir in der Mitte der Liste beginnen, und feststellen, ob der gesuchte Name vor oder hinter dieser Stelle steht. Liegt er dahinter, dann berücksichtigen wir die erste Hälfte nicht weiter und schauen das mittlere Element in der zweiten Hälfte an. Wir vergleichen diese Eintragung mit der gesuchten und beschränken die weitere Suche auf eine der beiden Hälften, usw. Die maximale Dauer der Suche geht mit log n, wobei n die Zahl der Elemente in der Tabelle ist.

Es gibt viele weitere Suchverfahren.

Zusammenfassung

Aufgabe dieses Abschnitts war es nur, gebräuchliche Datenstrukturen, die der Programmierer verwenden kann, kurz vorzustellen. Zwar wurden die allgemeinen Datenstrukturen zu Typen zusammengefaßt und mit Namen versehen, die gesamte Organisation der Daten in einem komplexen System kann jedoch irgendeine Kombination verwenden oder den Programmierer zwingen, passendere Strukturen selbst zu erfinden. Das Spektrum der Möglichkeiten wird nur von der Phantasie des Programmierers begrenzt. Ähnlich wurde eine große Zahl bekannter Such- und Sortierverfahren entwickelt, die mit den gebräuchlichen Datenstrukturen arbeiten. Eine zusammenfassende Beschreibung würde den Rahmen dieses Buchs sprengen. Dieser Abschnitt sollte zeigen, wie wichtig der Entwurf passender Strukturen für die Daten ist, die bearbeitet werden sollen, und er sollte die Grundwerkzeuge dazu liefern.

Tatsächliche Beispielprogramme werden jetzt genau vorgestellt.

2. Teil – Beispiele zum Entwurf

Einführung

Hier werden Beispiele für Anwendungen der typischen Datenstrukturen Tabelle, sortierte Liste und verkettete Liste, vorgestellt.

Praktische Algorithmen zur Suche, zum Einfügen und Löschen werden für diese Strukturen programmiert.

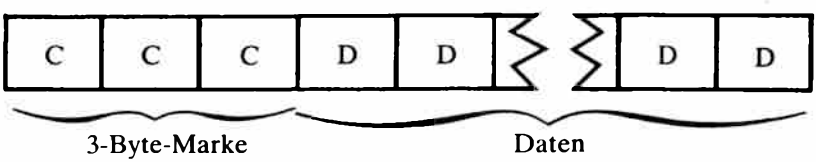
Der Leser, der sich für diese hochentwickelten Programmiertechniken interessiert, sei aufgefordert, die vorgestellten Programme genau zu untersuchen. Der Anfänger kann diesen Abschnitt zunächst überspringen und dann wieder darauf zurückkommen, wenn er sich dazu in der Lage fühlt.

Wenn man den Beispielen folgen will, dann muß man die Konzepte, die im ersten Abschnitt vorgestellt wurden, gut verstanden haben. Außerdem verwenden die Programme alle Adressierungsarten des Z80, und sie enthalten viele der Konzepte und Techniken, die in den vorhergehenden Kapiteln vorgestellt wurden.

Jetzt werden drei Strukturen eingeführt: eine einfache Liste, eine alphabetische Liste und eine verkettete Liste mit Inhaltsverzeichnis. Für jede Struktur werden drei Programme entwickelt, zum Suchen, Einfügen und Löschen.

Darstellung der Daten in der Liste

Sowohl die einfache Liste als auch die alphabetische Liste verwenden eine einheitliche Darstellung für jedes Element:



Jedes Element oder jede „Eintragung" enthält eine Marke und einen Block aus n Daten, wobei n zwischen 1 und 256 liegt. Deshalb belegt jede Eintragung maximal eine Seite (256 Byte). Innerhalb jeder Liste sind alle Elemente gleich lang (siehe Abb. 9.10). Die Programme, die mit

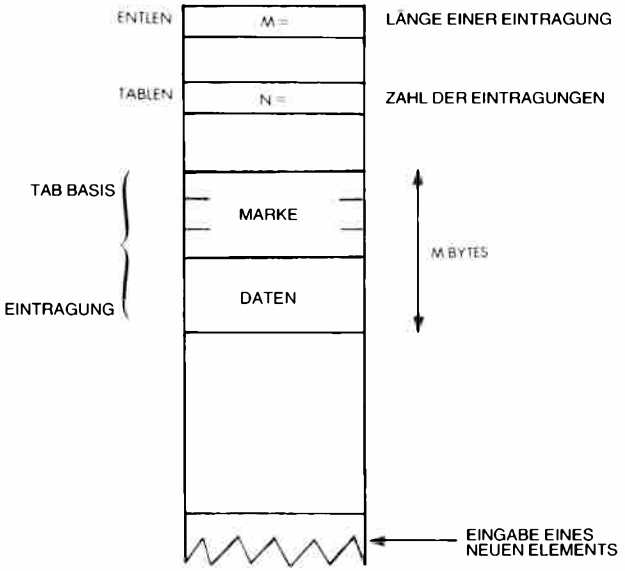


Abb. 9.9: Die Tabellenstruktur

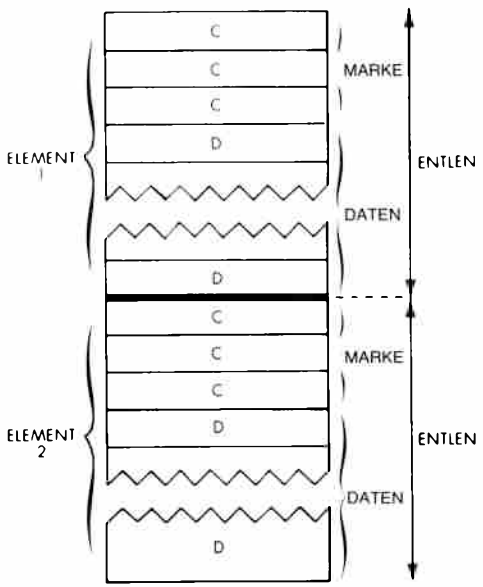


Abb. 9.10: Typische Eintragungen in die Liste

diesen beiden einfachen Listen arbeiten, verwenden einige gemeinsam festgelegte Variablen:

ENTLEN ist die Länge eines Elements. Enthält jedes Element beispielsweise 10 Byte Daten, dann ist ENTLEN=3+10=13
TABASE ist der Anfang der Tabelle oder Liste im Speicher
POINTER ist der Zeiger auf das aktuelle Element
OBJECT ist das aktuelle Element, das gesucht, eingetragen oder gelöscht werden soll
TABLEN ist die Zahl der Eintragungen.

Es wurde angenommen, daß alle Marken unterschiedlich sind. Will man diese Vereinbarung ändern, sind kleinere Änderungen in den Programmen notwendig.

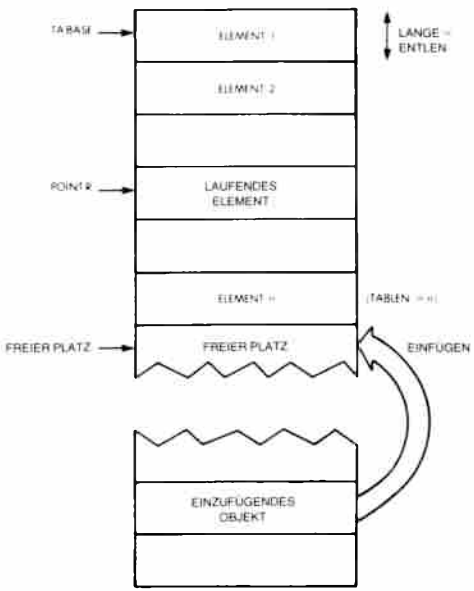


Abb. 9.11: Die einfache Liste

Eine einfache Liste

Die einfache Liste ist als Tabelle von n Elementen organisiert. Die Elemente sind nicht sortiert (siehe Abb. 9.11). Beim Suchen muß man die Liste von Anfang an durchgehen, bis man die Eintragung findet oder das Ende der Tabelle erreicht. Beim Einfügen werden neue Eintragungen an die bestehenden angehängt. Wird eine Eintragung gelöscht, werden eventuelle Eintragungen dahinter verschoben, um die Tabelle zusammenhängend zu halten.

Suchen

Es wird ein serielles Suchverfahren angewendet. Der Reihe nach wird das Markenfeld jeder Eintragung mit der Marke von OBJECT Buchstabe für Buchstabe verglichen. Der laufende Zeiger POINTR wird auf den Wert TABASE initialisiert.

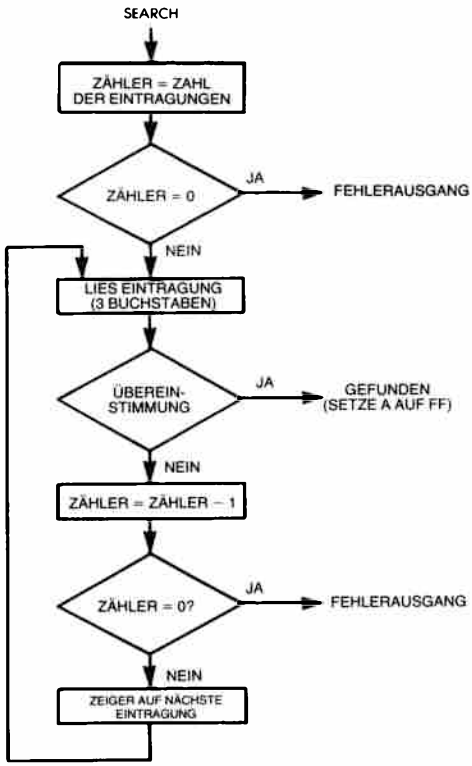


Abb. 9.12: Flußdiagramm zur Suche in der Tabelle

Die Suche wird auf naheliegende Art durchgeführt, Abbildung 9.12 zeigt das Flußdiagramm dazu. Das Suchprogramm erscheint in Abb. 9.16 am Ende des Abschnitts (das Programm „SEARCH"). Abbildung 9.17 zeigt einen Lauf des Programms als Beispiel.

Einfügen

Wird ein neues Element eingfügt, verwendet man dazu den ersten freien Block von (ENTLEN) Bytes am Ende der Liste (siehe Abb. 9.11).

Das Programm testet zuerst, ob die neue Eintragung nicht schon in der Liste vorhanden ist (in diesem Beispiel sollen alle Marken verschieden sein). Wenn nicht, dann wird die Listenlänge TABLEN inkrementiert

und OBJECT ans Ende der Liste angehängt. Das entsprechende Flußdiagramm zeigt Abb. 9.13.
Das Programm wird in Abb. 9.16 gezeigt. Es heißt „NEW" und belegt die Speicherzellen 0135 bis 015E.
Das Indexregister IY zeigt auf die Quelle, HL und DE auf das Ziel.

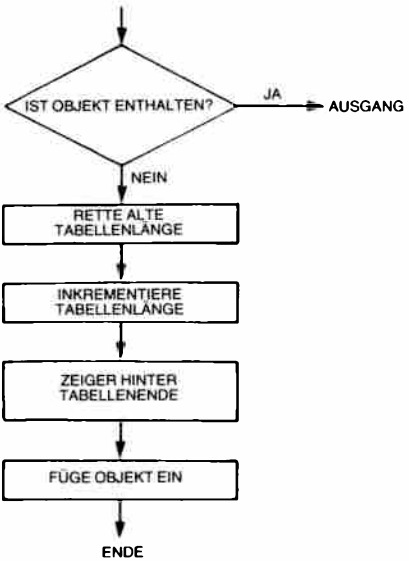


Abb. 9.13: Flußdiagramm zum Einfügen in die Tabelle

Löschen

Um ein Element aus der Liste zu löschen, werden die nachfolgenden Elemente bei den höheren Adressen um eine Elementposition nach unten verschoben. Dies ist in Abb. 9.14 dargestellt.

Das entsprechende Programm ist einfach und es erscheint in Abb. 9.16. Es wurde „DELETE" genannt und steht im Adreßbereich 015F bis 0187 im Speicher. Abb. 9.15 zeigt das Flußdiagramm.

Die Speicherstelle TEMPTR dient als Zeiger auf das aktuelle Element, das verschoben werden soll. Während des Transfers zeigt „POINTR" immer auf das „Loch" in der Liste, d. h. auf das Ziel des nächsten Blocktransfers.

Am Ausgang zeigt das Flag Z an, daß das Löschen erfolgreich ausgeführt wurde. Beachten Sie, daß der Befehl LDIR zum effizienten Blocktransfer benutzt wird (siehe Adresse 0178 in Abb. 9.16).

```
          LD    A,B              Blockzähler
NEWBLOC   LD    BC,(ENTLEN)      Blocklänge
          LDIR
          DEC   A
          JP    NZ,NEWBLOC
```

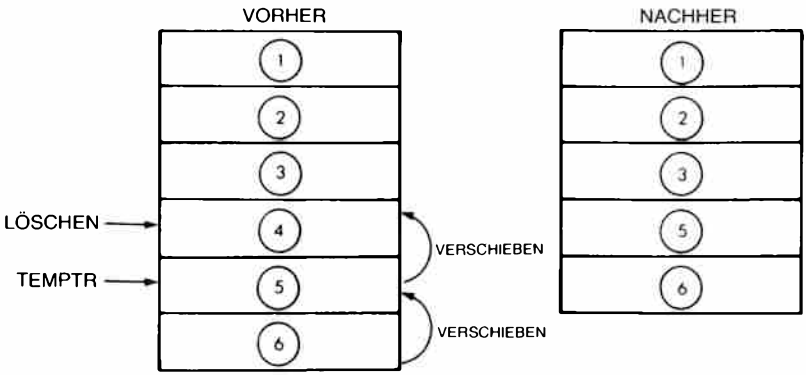


Abb. 9.14: Löschen einer Eintragung (einfache Liste)

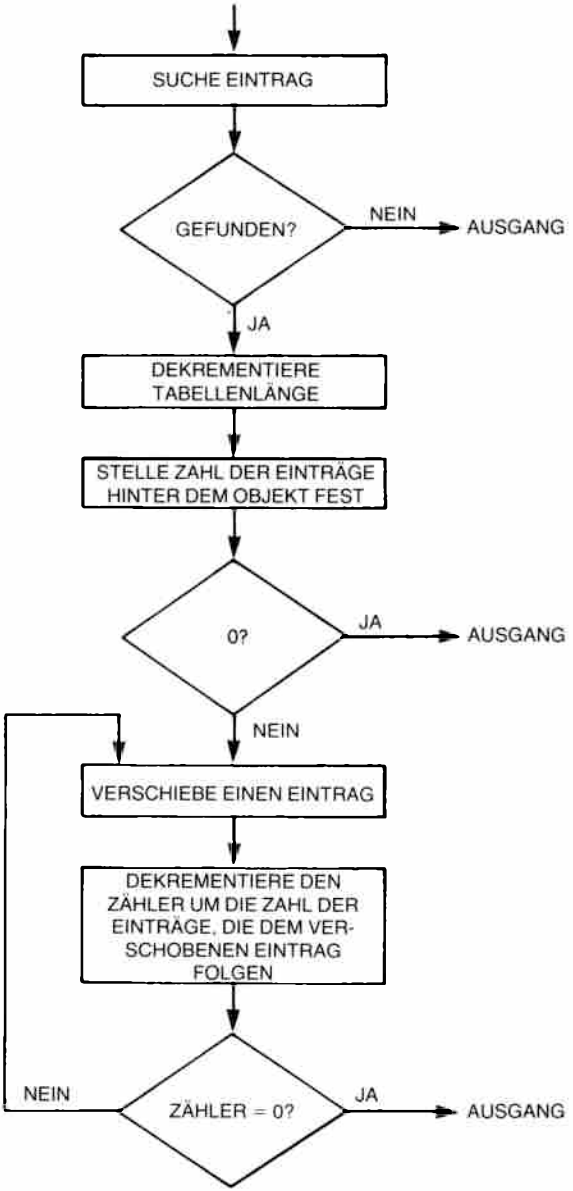


Abb. 9.15: Löschen in der Tabelle – Flußdiagramm

```
0000                    ORG   0100H
      (0187)    ENTLEN  DL    ENDER
      (0189)    TABLEN  DL    ENDER+2
      (018A)    TABASE  DL    ENDER+3
      (018C)    TEMP    DL    ENDER+5
                ;
0100  1600      SEARCH  LD    D,0             ;CLEAR D
0102  3A8901            LD    A,(TABLEN)      ;CHECK FOR A ZERO TABLE LENGTH
0105  A7                AND   A               ;SET FLAGS
0106  C8                RET   Z
0107  47                LD    B,A             ;STORE TABLE LENGTH
0108  DD2A8A01          LD    IX,(TABASE)     ;PUT BASE ADDR. IN IX
010C  DD7E00    LOOP    LD    A,(IX+0)        ;CHECK FIRST LETTER OF ENTRY
010F  FDBE00            CP    (IY+0)
0112  C22701            JP    NZ,NEXTONE
0115  DD7E01            LD    A,(IX+1)        ;CHECK 2ND LETTER
0118  FDBE01            CP    (IY+1)
011B  C22701            JP    NZ,NEXTONE
011E  DD7E02            LD    A,(IX+2)        ;CHECK 3RD LETTER
0121  FDBE02            CP    (IY+2)
0124  CA3201            JP    Z,FOUND         ;EXIT IF ALL LETTERS MATCH
0127  05        NEXTONE DEC   B               ;DECREMENT TABLE LENGTH COUNTER
0128  C8                RET   Z               ;EXIT IF AT END OF TABLE
0129  ED5B8701          LD    DE,(ENTLEN)     ;SET IX TO NEXT ENTRY ADDR.
012D  DD19              ADD   IX,DE
012F  C30C01            JP    LOOP            ;TRY AGAIN
0132  16FF      FOUND   LD    D,0FFH          ;SET D TO SHOW IX CONTAINS ADDR.
0134  C9                RET                   ;..OF ENTRY IN TABLE
                ;
                ;
                ;
0135  CD0001    NEW     CALL  SEARCH          ;SEE IF OBJECT IS THERE
0138  14                INC   D
0139  CA5E01            JP    Z,OUTE          ;IF D WAS FF, EXIT
013C  3A8901            LD    A,(TABLEN)
013F  5F                LD    E,A             ;LOAD E WITH TABLE LENGTH
0140  3C                INC   A
0141  328901            LD    (TABLEN),A      ;INCREMENT TABLE LENGTH
0144  1600              LD    D,0
0146  2A8A01            LD    HL,(TABASE)
0149  ED4B8701          LD    BC,(ENTLEN)     ;SET B TO LENGTH OF AN ENTRY
014D  41                LD    B,C
014E  19        LOOPE   ADD   HL,DE
014F  10FD              DJNZ  LOOPE           ;ADD HL TO (ENTLEN)*TABLEN)
0151  ED4B8701          LD    BC,(ENTLEN)
0155  FDE5              PUSH  IY              ;MOVE IY TO DE
0157  D1                POP   DE
0158  EB                EX    DE,HL
0159  EDB0              LDIR                  ;MOVE MEMORY FROM OBJECT TO END
015B  01FFFF            LD    BC,0FFFFH       ;..OF TABLE
015E  C9        OUTE    RET
                ;
                ;
                ;
015F  CD0001    DELETE  CALL  SEARCH          ;FIND ENTRY TO BE DELETED
0162  14                INC   D               ;SEE IF IT WAS FOUND
0163  C28601            JP    NZ,OUT
0166  3A8901            LD    A,(TABLEN)      ;DECREMENT TABLE LENGTH
0169  3D                DEC   A
016A  328901            LD    (TABLEN),A
016D  05                DEC   B               ;B NOW=# OF ENTRIES LEFT IN TABLE
016E  CA8301            JP    Z,EXIT          ;..AFTER ONE TO BE DELETED
0171  DDE5              PUSH  IX              ;MOVE IX TO DE
0173  D1                POP   DE
0174  2A8701            LD    HL,(ENTLEN)     ;SET HL ONE ENTRY AHEAD OF DE
0177  19                ADD   HL,DE
0178  78                LD    A,B             ;SET BLOCK COUNTER
0179  ED4B8701  NEWBLOC LD    BC,(ENTLEN)     ;SET BLOCK LENGTH COUNTER
017D  EDB0              LDIR                  ;SHIFT 1 ENTRY OF TABLE
017F  3D                DEC   A
0180  C27901            JP    NZ,NEWBLOC      ;SHIFT ANOTHER BLOCK
0183  01FFFF    EXIT    LD    BC,0FFFFH       ;SHOW THAT IT WAS DONE
0186  C9        OUT     RET
                ;
0187  (0000)    ENDER   END
```

Abb. 9.16: Einfache Liste – die Programme

```
SYMBOL TABLE

DELETE  015F    ENDER   0187    ENTLEN  0187    EXIT    0183    FOUND   0132
LOOP    010C    LOOPE   014E    NEW     0135    NEWBLO  0179    NEXTON  0127
OUT     0186    OUTE    015E    SEARCH  0100    TABASE  018A    TABLEN  0189
TEMP    018C
```

Abb. 9.16: Einfache Liste – die Programme (Fortsetzung)

Anzeige des Speicherinhalts

Liste der Objekte mit ihren Adressen im Speicher

```
-DM300
0300  53 4F 4E 31 31 31 31 31-31 31 31 31 31 00 00 00  SON1111111111...
0310  44 41 44 32 32 32 32 32-32 32 32 32 32 00 00 00  DAD2222222222...
0320  4D 4F 4D 33 33 33 33 33-33 33 33 33 33 00 00 00  MOM3333333333...
0330  55 4E 43 34 34 34 34 34-34 34 34 34 34 00 00 00  UNC4444444444...
0340  41 4E 54 35 35 35 35 35-35 35 35 35 35 00 00 00  ANT5555555555...
0350  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0360  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0370  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
```

```
-SY
Y=0000 300
```

Setze IY auf 0300H (Zeiger auf Objekt)

```
-G193/196
```

```
P=0196 0196'
```

Starte 'INSERT'

Tabelle nach Start des Programms

```
-DM400
0400  53 4F 4E 31 31 31 31 31-31 31 31 31 31 00 00 00  SON1111111111...
0410  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0420  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0430  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0440  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0450  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0460  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0470  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
```

```
-SY
Y=0300 310
```

Setze IY auf 0310H (nächstes Objekt)

```
-G193/196
```

```
P=0196 0196'
```

Starte 'INSERT'

Tabelle nach dem zweiten Einfügen

```
-DM400
0400  53 4F 4E 31 31 31 31 31-31 31 31 31 31 44 41 44  SON1111111111DAD
0410  32 32 32 32 32 32 32 32-32 32 00 00 00 00 00 00  2222222222......
0420  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0430  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0440  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0450  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0460  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0470  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
```

*** *(weitere Einfügungen)* ***

Tabelle nach mehreren Einfügungen

```
-DM400
0400  53 4F 4E 31 31 31 31 31-31 31 31 31 31 44 41 44  SON1111111111DAD
0410  32 32 32 32 32 32 32 32-32 32 55 4E 43 34 34 34  2222222222UNC444
0420  34 34 34 34 34 34 34 34-4D 4F 4D 33 33 33 33 33  4444444MOM333333
0430  33 33 33 33 41 4E 54 35-35 35 35 35 35 35 35 35  3333ANT555555555
0440  35 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  5...............
0450  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0460  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0470  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
```

Abb.9.17: Einfache Liste – ein Durchlauf als Beispiel

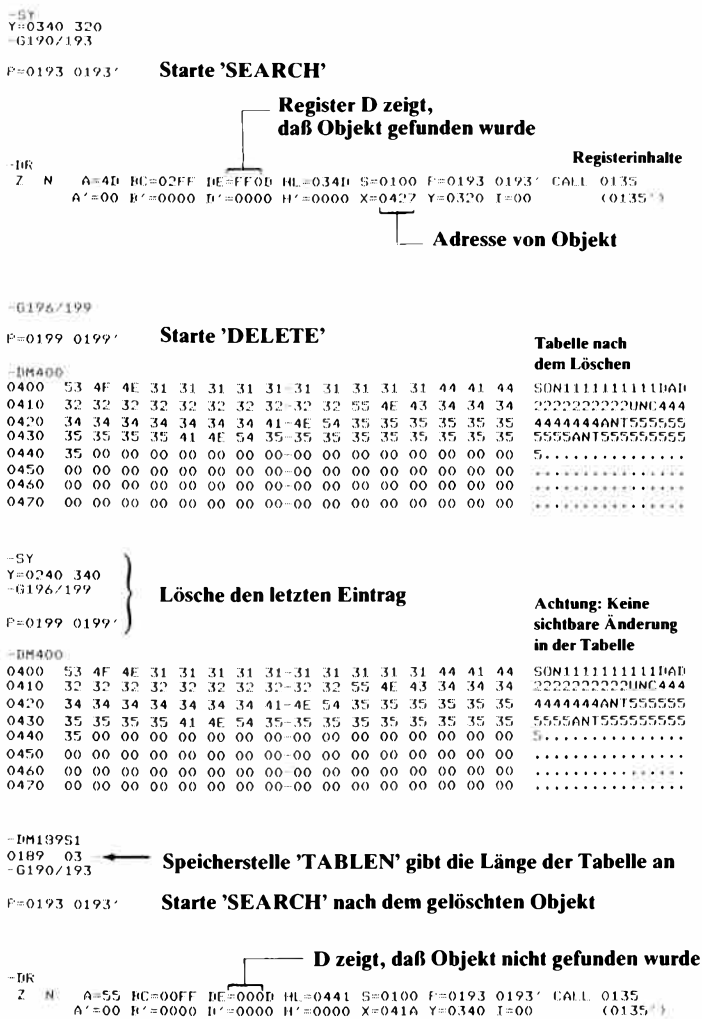


Abb. 9.17: Einfache Liste – ein Durchlauf als Beispiel (Fortsetzung)

Alphabetische Liste

Im Gegensatz zum letzten Beispiel enthält die alphabetische Liste alle Elemente in alphabetischer Reihenfolge. Dies erlaubt die Verwendung schnellerer Suchverfahren als im vorhergehenden Beispiel. Hier wird ein binäres Suchverfahren verwendet.

Suche

Der Suchalgorithmus ist ein klassisches binäres Suchen. Wir wollen uns daran erinnern, daß diese Technik im wesentlichen analog zu dem Verfahren ist, nach dem man einen Namen im Telefonbuch sucht. Man beginnt normalerweise irgendwo in der Mitte des Buchs und geht dann nach vorne oder nach hinten, abhängig von den Eintragungen, die man dort findet. Diese Methode ist schnell und kann relativ einfach eingebaut werden.

Das Flußdiagramm zur binären Suche zeigt Abb. 9.18, das Programm Abb. 9.23.

Die Liste enthält die Eintragungen in alphabetischer Reihenfolge und greift durch binäres oder „logarithmisches" Suchen auf sie zu. Abb. 9.19 zeigt ein Beispiel. Die Suche wird dadurch etwas erschwert, daß man verschiedene Bedingungen beachten muß. Das Hauptproblem ist es, zu vermeiden, daß man ein Element sucht, das es nicht gibt. In einem solchen Fall könnte man die Eintragungen mit unmittelbar höheren oder tieferen alphabetischen Werten ewig testen. Um dies zu verhindern, enthält das Programm ein Flag, in dem der Wert des Carryflags nach einem erfolglosen Vergleich gespeichert wird. Sobald INCMNT, das angibt, um wieviel der Zeiger beim nächsten Mal inkrementiert wird, den Wert „1" erreicht, wird ein anderes Flag namens „CLOSENOW" (abgekürzt „CLOSE") auf den WERT des Flags COMPRES gesetzt. Da ab dann jeweils um „1" inkrementiert wird, wird COMPRES verschieden von CLOSE, falls der Zeiger hinter die Stelle gelangt, wo das Element liegen sollte. Dann wird die Suche abgebrochen. Damit hat auch die Routine NEW die Möglichkeit festzustellen, wohin die logischen und physikalischen Zeiger zeigen und wohin das neue Element kommt.

Wenn das gesuchte Element nicht in der Tabelle steht und der Zeiger um Eins inkrementiert wird, wird das Flag CLOSE gesetzt. Im nächsten Durchgang durch das Programm ist das Ergebnis des Vergleichs umgekehrt wie vorher. Die beiden Flags stimmen nicht mehr überein und das Programm endet mit der Anzeige „nicht gefunden".

Das andere Hauptproblem, mit dem man sich befassen muß, ist die Möglichkeit, daß man über ein Ende der Tabelle hinausgerät, wenn man das Inkrement addiert oder subtrahiert. Dies wird dadurch gelöst, indem man eine Testaddition oder -subtraktion mit den logischen Zeigern und der Länge durchführt, die die tatsächliche Zahl der Eintragungen, nicht die physikalische Position im Speicher angeben.

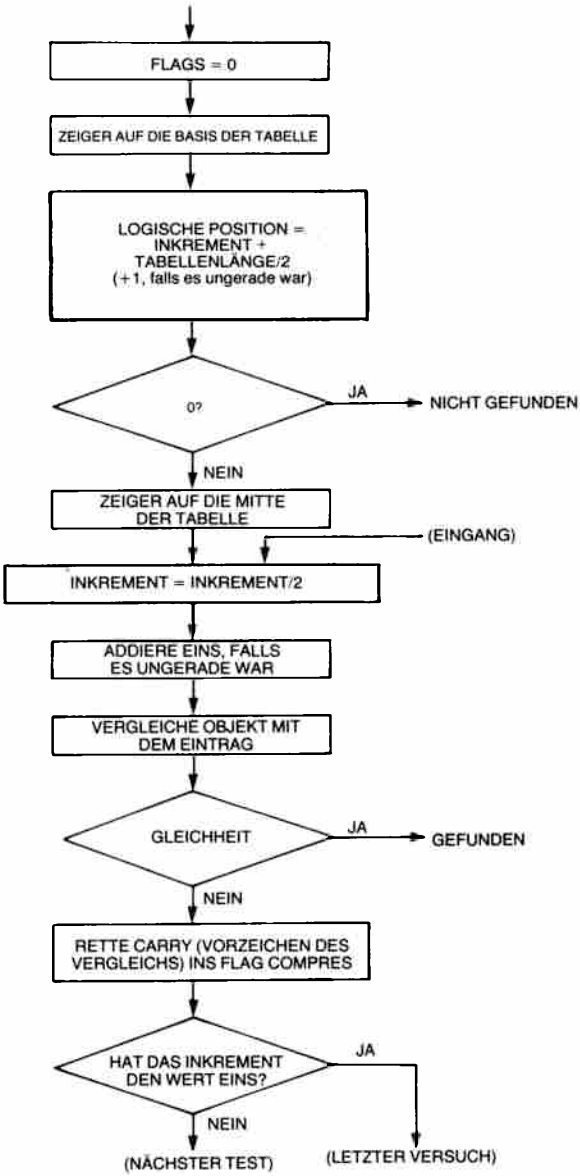
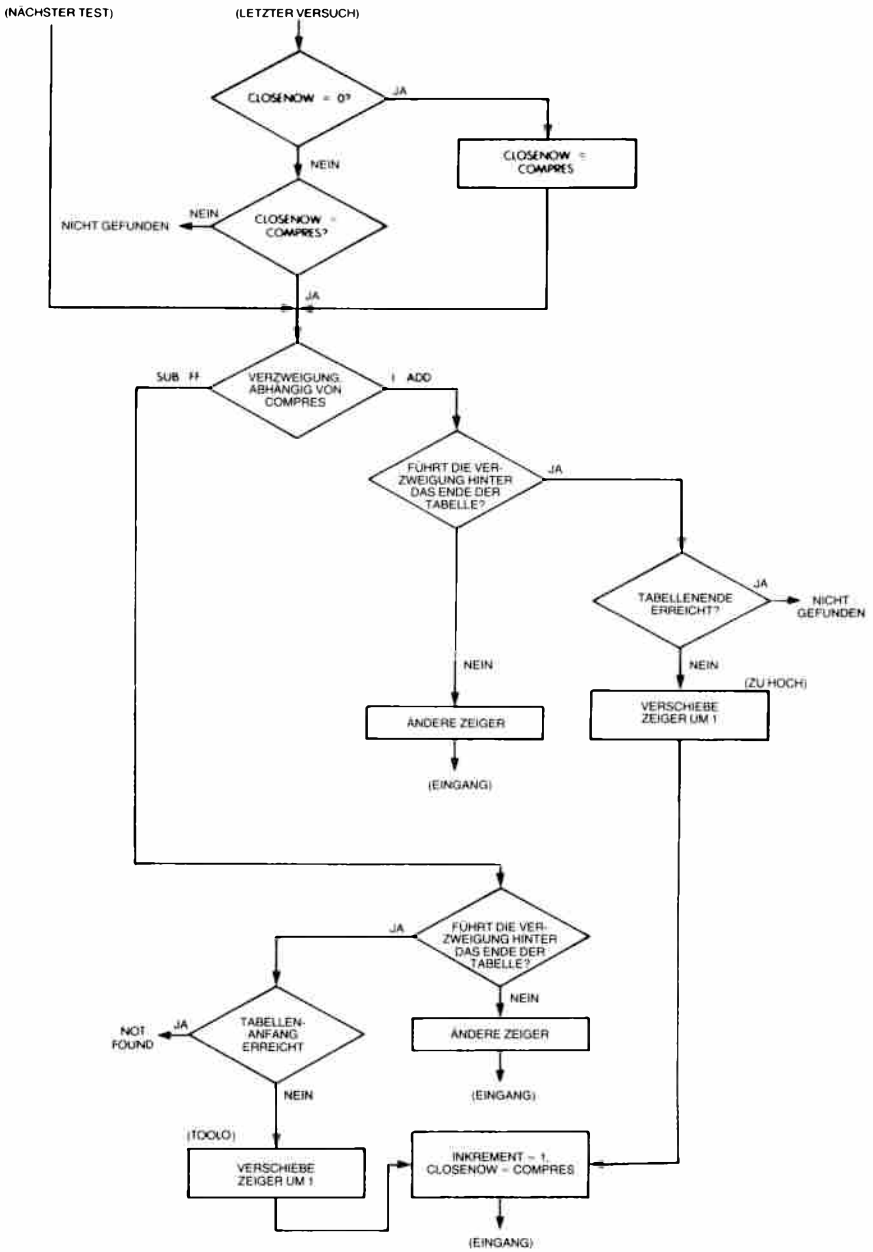


Abb. 9.18: Binäres Suchen – Flußdiagramm


(NÄCHSTER TEST)
(LETZTER VERSUCH)
CLOSENOW = 0?
JA
NEIN
CLOSENOW = COMPRES
NEIN
NICHT GEFUNDEN
CLOSENOW = COMPRES?
JA
SUB ff
VERZWEIGUNG ABHÄNGIG VON COMPRES
1 ADD
FÜHRT DIE VERZWEIGUNG HINTER DAS ENDE DER TABELLE?
JA
NEIN
TABELLENENDE ERREICHT?
JA
NICHT GEFUNDEN
NEIN
(ZU HOCH)
ÄNDERE ZEIGER
VERSCHIEBE ZEIGER UM 1
(EINGANG)
JA
FÜHRT DIE VERZWEIGUNG HINTER DAS ENDE DER TABELLE?
NEIN
NOT FOUND
JA
TABELLENANFANG ERREICHT
ÄNDERE ZEIGER
(EINGANG)
NEIN
(TOOLO)
VERSCHIEBE ZEIGER UM 1
INKREMENT = 1, CLOSENOW = COMPRES
(EINGANG)

```
(0121)      LD      A,  C
            SRL     A
            ADC     0
            LD      C,  A
```

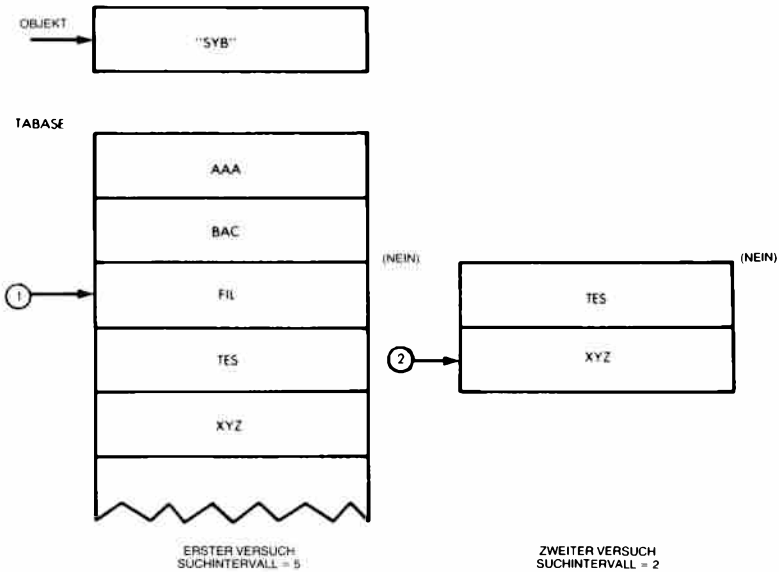


Abb. 9.19: Eine binäre Suche

Zusammengefaßt verwendet das Programm die beiden Flags COMPRES und CLOSE, um Information zu speichern. Das Flag COMPRES gibt an, ob der Übertrag beim letzten Vergleich „0" oder „1" war. Dies gibt an, ob das verglichene Element größer oder kleiner war, als das, mit dem es verglichen wurde. War der Übertrag C „1" und das Element kleiner, dann wird COMPRES auf „1" gesetzt. War C „0", was anzeigt, daß das Element größer war, wird COMPRES auf „FF" gesetzt.

Als zweites Flag verwendet das Programm CLOSE. Wenn das Suchinkrement INCMNT „1" wird, ist dieses Flag gleich COMPRES. Es zeigt beim nächsten Durchgang an, daß das Element nicht gefunden wurde, wenn COMPRES nicht gleich CLOSE ist. Das Programm verwendet außerdem folgende Variablen:

LOGPOS gibt die logische Position in der Tabelle an (Elementnr)
INCMNT gibt an, um wieviel der laufende Zeiger beim nächsten Mal inkrementiert oder dekrementiert wird, wenn der Vergleich fehlschlägt
TABLEN gibt wie üblich die gesamte Länge der Tabelle an.

LOGPOS und INCMNT werden mit TABLEN verglichen, um sicherzustellen, daß die Grenzen der Liste nicht überschritten werden.

Das Programm mit dem Namen „SEARCH" ist in Abb. 9.23 dargestellt. Es belegt den Speicherbereich 0100 bis 01CF und es ist es Wert, sorgfältig studiert zu werden, da es erheblich komplizierter ist als die lineare Suche.

Eine zusätzliche Schwierigkeit besteht darin, daß die Suchintervalle jeweils eine gerade oder eine ungerade Zahl von Elementen enthalten können. Ist sie ungerade, muß eine Korrektur durchgeführt werden. (Man kann z. B. nicht auf das mittlere Element einer Tabelle aus vier Elementen zeigen.) Dann dient ein „Trick" dazu, auf das mittlere Element zu zeigen: Die Division durch 2 wird durch ein Rechts-Schieben ausgeführt. Das Bit, das nach dem Befehl SRL ins Carry „herausfällt", ist eine „1", wenn das Intervall ungerade war. Es wird einfach zum Zeiger addiert.

Das OBJECT wird dann mit der Eintragung in der Mitte des neuen Suchintervalls verglichen. Ist der Vergleich erfolgreich, wird das Programm beendet. Ansonsten („NOGOOD") wird das Carry auf „0" gesetzt, wenn OBJECT kleiner ist als die Eintragung. Wenn INCMNT „1" wird, dann wird das Flag CLOSE (das anfangs auf Null gesetzt war) getestet, um zu sehen, ob es gesetzt war. Falls nicht, dann wird es gesetzt. War es gesetzt, dann wird getestet, ob die Stelle bereits passiert ist, an der OBJECT hätte stehen sollen.

Beachten Sie auch, daß der laufende Zeiger auf die Eintragung unterhalb von OBJECT zeigt, wenn das Carry „1" war.

Einfügen eines Element

Um ein neues Element einzufügen, wird zunächst eine binäre Suche ausgeführt. Wird das Element in der Tabelle gefunden, dann braucht es nicht mehr eingefügt zu werden. (Wir nehmen hier an, daß alle Elemente verschieden sind.) Wurde das Element in der Tabelle nicht gefunden, muß es direkt vor oder direkt hinter dem Element eingefügt werden, mit dem es als letztes verglichen wurde. Nach der Suche gibt der Wert des Flags COMPRES an, ob es davor oder dahinter eingefügt werden muß. Alle Elemente hinter der Stelle, an der es eingefügt werden soll, werden um einen Block nach hinten verschoben und das neue Element wird eingefügt.

In Abb. 9.20 ist der Vorgang des Einfügens veranschaulicht, das entsprechende Programm erscheint in Abb. 9.23.

Das Programm heißt NEW und beginnt bei der Adresse 01D0. Beachten Sie, daß die automatischen Z80-Blocktransferbefehle LDDR und LDIR für einen effizienten Blocktransfer verwendet werden.

Löschen eines Elements

Es wird entsprechend eine binäre Suche ausgeführt, um das Element zu finden. Schlägt die Suche fehl, dann braucht es nicht gelöscht zu werden.

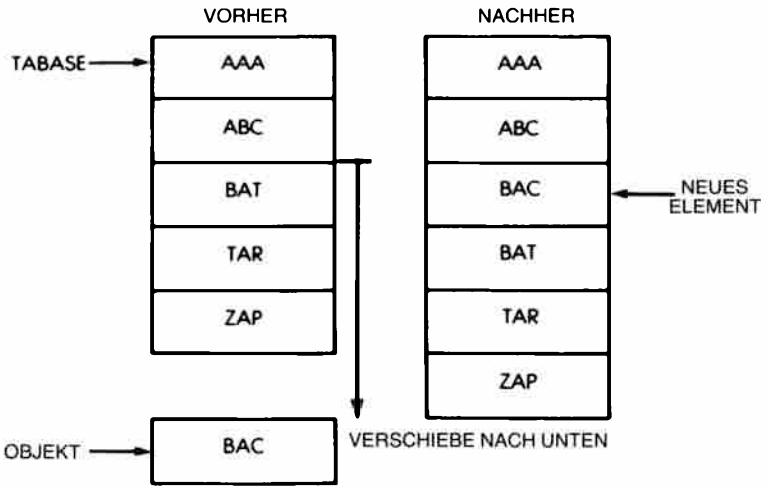


Abb. 9.20: Füge „BAC" ein

Ist die Suche erfolgreich, dann wird das Element gelöscht und alle folgenden Elemente um eine Blockposition nach oben verschoben. Abb. 9.21 zeigt ein entsprechendes Beispiel, und das Programm erscheint in Abb. 9.23. Abbildung 9.22 enthält das zugehörige Flußdiagramm.

Das Programm heißt DELETE und beginnt bei der Adresse 0221. In Abb. 9.24 ist ein Lauf des Programms als Beispiel dargestellt.

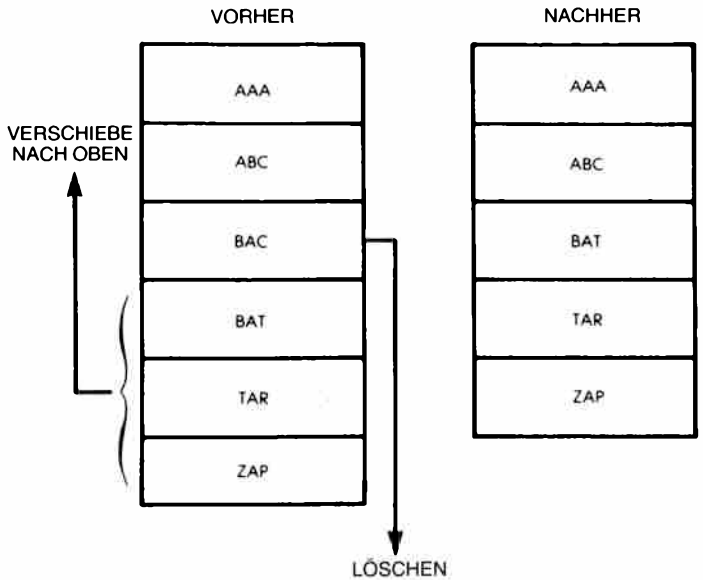


Abb. 9.21: Lösche „BAC"

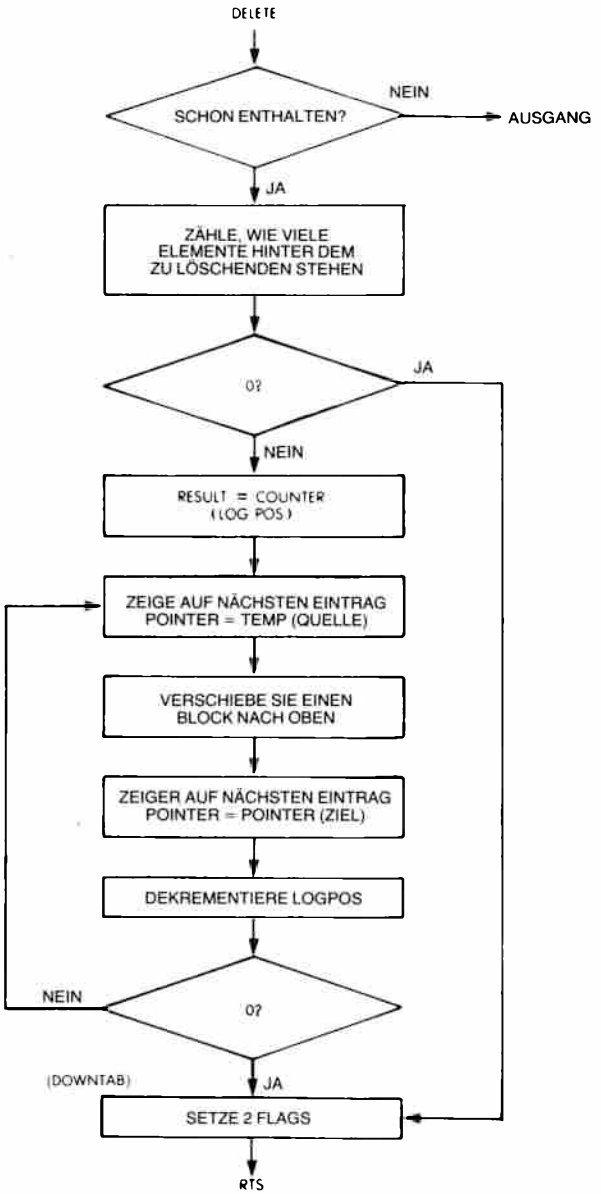


Abb. 9.22: Flußdiagramm zum Löschen (alphabetische Liste)

```
0000                        ORG   0100H
      (024A)    CLOSENOW    DL    ENDED
      (024B)    COMPRES     DL    ENDED+1
      (024C)    TABLEN      DL    ENDED+2
      (024D)    TABASE      DL    ENDED+3
      (024F)    ENTLEN      DL    ENDED+5
                ;
0100  3E00      SEARCH      LD    A,0
0102  324A02                LD    (CLOSENOW),A    ;ZERO FLAG LOCATIONS
0105  324B02                LD    (COMPRES),A
0108  57                    LD    D,A
0109  2A4D02                LD    HL,(TABASE)     ;INITIALIZE HL
010C  3A4C02                LD    A,(TABLEN)
010F  CB3F                  SRL   A               ;DIVIDE BY 2
0111  CE00                  ADC   0               ;ADD 1'S BIT BACK IN
0113  4F                    LD    C,A             ;STORE AS INCREMENT VALUE
0114  47                    LD    B,A             ;STORE AS LOGICAL POSITION VALUE
0115  CABA01                JP    Z,NOTFOUND      ;CHECK IF LENGTH IS ZERO
0118  5F                    LD    E,A             ;MULTIPLY (E=1)*ENTLEN
0119  1D                    DEC   E
011A  CDBD01                CALL  MULT
011D  19                    ADD   HL,DE           ;SET HL TO MIDDLE OF TABLE
011E  E5          ENTRY     PUSH  HL              ;LOAD HL INTO IX
011F  DDE1                  POP   IX
0121  79                    LD    A,C             ;DIVIDE INCREMENT VALUE BY TWO
0122  CB3F                  SRL   A
0124  CE00                  ADC   0
0126  4F                    LD    C,A
0127  DD7E00                LD    A,(IX+0)        ;COMPARE FIRST LETTER
012A  FDBE00                CP    (IY+0)
012D  C24201                JP    NZ,NOGOOD
0130  DD7F01                LD    A,(IX+1)        ;COMPARE 2ND LETTER
0133  FDBE01                CP    (IY+1)
0136  C24201                JP    NZ,NOGOOD
0139  DD7E02                LD    A,(IX+2)        ;COMPARE 3RD LETTER
013C  FDBE02                CP    (IY+2)
013F  CABC01                JP    Z,FOUND
0142  3F01      NOGOOD      LD    A,1             ;SET COMPARE RESULT FLAG TO
0144  DA4901                JP    C,TESTS         ;..RESULT OF COMPARE (1,FF)
0147  3EFF                  LD    A,0FFH
0149  324B02    TESTS       LD    (COMPRES),A
014C  79                    LD    A,C             ;IS INCREMENT VALUE 1?
014D  3D                    DEC   A
014E  C26901                JP    NZ,NEXTEST
0151  3A4A02                LD    A,(CLOSENOW)    ;YES, IS CLOSE FLAG SET?
0154  A7                    AND   A
0155  CA6301                JP    Z,NOTCLOSE
0158  57                    LD    D,A             ;YES,SEE IF HAVE PASSED WHERE
0159  3A4B02                LD    A,(COMPRES)     ;..ENTRY SHOULD BE BUT ISN'T
015C  92                    SUB   D
015D  CA6901                JP    Z,NEXTEST
0160  C3BA01                JP    NOTFOUND
0163  3A4B02    NOTCLOSE    LD    A,(COMPRES)     ;SET CLOSE FLAG TO DIRECTION OF
0166  324A02                LD    (CLOSENOW),A    ;..SEARCH TO PREVENT REPETITION
0169  DDE5      NEXTEST     PUSH  IX              ;PREPARE HL AND DE FOR ADD OR
016B  F1                    POP   HL              ;..SUB OF INCREMENT VALUE
016C  59                    LD    E,C
016D  CDBD01                CALL  MULT
0170  3A4B02                LD    A,(COMPRES)     ;TEST IF WANT TO ADD OR SUB
0173  3C                    INC   A
0174  C29601                JP    NZ,ADDIT
0177  78                    LD    A,B             ;TEST TO SEE IF SUB WILL RUN
0178  91                    SUB   C               ;..OFF BOTTOM OF TABLE
0179  CA8501                JP    Z,TOOLOW
017C  DA8501                JP    C,TOOLOW
017F  47                    LD    B,A             ;SET NEW LOGICAL POSITION VALUE
0180  ED52                  SBC   HL,DE           ;CHANGE ADDRESS ITSELF
0182  C31F01                JP    ENTRY
0185  78        TOOLOW      LD    A,B             ;SEE IF POSITION IS 1
0186  3D                    DEC   A
0187  CABA01                JP    Z,NOTFOUND      ;IF SO, EXIT
018A  ED5B4F02              LD    DE,(ENTLEN)     ;JUST SUB 1 ENTRY POSITION
018E  37                    SCF
018F  3F                    CCF
0190  ED52                  SBC   HL,DE
0192  05                    DEC   B               ;CHANGE LOGICAL POSITION
0193  C3AF01                JP    REALCLOS
```

Abb. 9.23: Binäres Suchprogramm

```
0196  3A4C02    ADDIT   LD    A,(TABLEN)      ;TEST TO SEE IF CURRENT POSITION
0199  90                 SUB   B               ;...PLUS INCREMENT WILL GO PAST
019A  91                 SUB   C               ;...END OF THE TABLE
019B  DAA501             JP    C,TOOHIGH
019E  19                 ADD   HL,DE           ;IS OK, CHANGE ACTUAL ADDRESS
019F  78                 LD    A,B             ;CHANGE LOGICAL POS. VALUE
01A0  81                 ADD   C
01A1  47                 LD    B,A
01A2  C31E01             JP    ENTRY
01A5  81          TOOHIGH ADD   C               ;SEE IF POSITION IS AT TOP OF
01A6  CABA01             JP    Z,NOTFOUND      ;...TABLE (SAME AS TABLEN-1)
01A9  ED5B4E02           LD    DE,(ENTLEN)     ;ADD 1 ENTRY POSITION
01AD  19                 ADD   HL,DE
01AE  04                 INC   B               ;INCREMENT LOGICAL POSITION
01AF  0E01        REALCLOS LD    C,1             ;SET INCREMENT TO 1
01B1  3A4B02             LD    A,(COMPRES)     ;SET CLOSE FLAG TO COMPARE
01B4  324A02             LD    (CLOSENOW),A    ;...RESULT
01B7  C31E01             JP    ENTRY
01BA  16FF        NOTFOUND LD    D,0FFH
01BC  C9          FOUND   RET
                  ;
01BD  E5          MULT    PUSH  HL              ;MULTIPLIES E BY (ENTLEN),
01BE  C5                  PUSH  BC              ;...VALUE IN DE ON EXIT
01BF  1600                LD    D,0
01C1  210000              LD    HL,0000
01C4  ED4B4E02            LD    BC,(ENTLEN)
01C8  41                  LD    B,C
01C9  19          ADDEM   ADD   HL,DE
01CA  10FD                DJNZ  ADDEM
01CC  C1                  POP   BC
01CD  EB                  EX    DE,HL
01CE  E1                  POP   HL
01CF  C9                  RET
                  ;
                  ;
                  ;
01D0  CD0001      NEW     CALL  SEARCH          ;SEE IF OBJECT IS ALREADY THERE
01D3  14                  INC   D
01D4  C22002              JP    NZ,QUIT
01D7  3A4C02              LD    A,(TABLEN)      ;CHECK FOR 0 TABLE
01DA  A7                  AND   A
01DB  CA0C02              JP    Z,INSERT
01DE  3A4B02              LD    A,(COMPRES)
01E1  3C                  INC   A
01E2  CAEB01              JP    Z,HITSIDE
01E5  ED5B4E02            LD    DE,(ENTLEN)     ;COMPRES 1, SET HL ABOVE WHERE
01E9  19                  ADD   HL,DE           ;...OBJECT SHOULD GO
01EA  C3EE01              JP    SETUP
01ED  05          HITSIDE DEC   B               ;COMPRES 0, SET B FOR SUBTRACT
01EE  3A4C02      SETUP   LD    A,(TABLEN)      ;SEE HOW MANY ENTRIES ARE LEFT
01F1  90                  SUB   B
01F2  CA0C02              JP    Z,INSERT
01F5  5F                  LD    E,A             ;SET HL TO LAST POSITION IN LAST
01F6  CDBB01              CALL  MULT            ;...ENTRY
01F9  19                  ADD   HL,DE
01FA  2B                  DEC   HL
01FB  EB                  EX    DE,HL           ;SET DE 1 ENTRY ABOVE HL
01FC  2A4E02              LD    HL,(ENTLEN)
01FF  19                  ADD   HL,DE
0200  EB                  EX    DE,HL
0201  ED4B4E02    MOVEM   LD    BC,(ENTLEN)     ;SHIFT UP ONE ENTRY OF MEMORY
0205  EDB8                LDDR
0207  3D                  DEC   A
0208  C20102              JP    NZ,MOVEM        ;REPEAT IF NECCESSARY
020B  23                  INC   HL              ;HL IS FRONT OF NOW EMPTY SPACE
020C  EDE5        INSERT  PUSH  IY              ;LOAD OBJECT INTO EMPTY SPACE
020E  D1                  POP   DE
020F  EB                  EX    DE,HL
0210  ED4B4E02            LD    BC,(ENTLEN)
0214  EDB0                LDIR
0216  3A4C02              LD    A,(TABLEN)      ;INCREMENT TABLE LENGTH
0219  3C                  INC   A
021A  324C02              LD    (TABLEN),A
021D  01FFFF              LD    BC,0FFFFH       ;SHOW THAT IT WAS DONE
0220  C9          QUIT    RET
                  ;
                  ;
                  ;
```

Abb. 9.23: Binäres Suchprogramm (Fortsetzung)

```
0221  CD0001    DELETE   CALL  SEARCH          ;GET ADDRESS OF OBJECT
0224  14                 INC   D               ;SEE IF OBJECT IS THERE
0225  CA4902             JP    Z,OUTE
0228  ED5B4F02           LD    DE,(ENTLEN)
022C  EB                 EX    DE,HL
022D  19                 ADD   HL,DE           ;DE IS LOC. OF OBJECT, HL IS
022E  3A4C02             LD    A,(TABLEN)      ;..ONE ENTRY ORDVE
0231  90                 SUB   B               ;SEE HOW MANY ENTRIES ARE LEFT
0232  CA3F02             JP    Z,DOWNTAB
0235  ED4B4F02  SHIFTIN  LD    BC,(ENTLEN)
0239  EDB0              LDIR                  ;SHIFT DOWN 1 ENTRY LENGTH
023B  3D                 DEC   A
023C  C23502             JP    NZ,SHIFTIN
023F  3A4C02    DOWNTAB  LD    A,(TABLEN)      ;DECREMENT TABLE LENGTH
0242  3D                 DEC   A
0243  324C02             LD    (TABLEN),A
0246  01FFFF             LD    BC,0FFFFH       ;SHOW THAT ACTION WAS TAKEN
0249  C9        OUTE     RET
                 ;
024A  (0000)     ENDED    END

SYMBOL TABLE

ADDEM   01C9   ADDIT   0196   CLOSEN  024A   COMPRE  024B   DELETE  0221
DOWNTA  023F   ENDED   024A   ENTLEN  024F   ENTRY   011E   FOUND   01BC
HISIDE  01ED   INSERT  020C   MOVEM   0201   MULT    01BD   NEW     01D0
NEXTES  0169   NOGOOD  0142   NOTCLO  0163   NOTFOU  01BA   OUT     0220
OUTE    0249   REALCL  01AF   SEARCH  0100   SETUP   01EE   SHIFTI  0235
TABASE  024D   TABLEN  024C   TESTS   0149   TOOHIG  01A5   TOOLOW  0185
```

Abb. 9.23: Binäres Suchprogramm (Fortsetzung)

Verkettete Liste

Die verkettete Liste soll wie üblich die drei alphanumerischen Zeichen für die Marke enthalten, gefolgt von einem bis 250 Datenbytes, einem Zeiger aus zwei Byte, der die Startadresse der nächsten Eintragung enthält, und einer Markierung aus einem Byte. Ist diese Markierung auf „1" gesetzt, dann wird das Einfügeprogramm davon abgehalten, diese Eintragung durch eine neue zu ersetzen.

Weiterhin enthält ein Inhaltsverzeichnis einen Zeiger auf die erste Eintragung für jeden Buchstaben des Alphabets, um den Zugriff zu erleichtern. Es wird vom Programm angenommen, daß die Marken ASCII-Zeichen sind. Alle Zeiger am Ende der Liste werden auf ein Null-Zeichen gesetzt, das hier gleich der Anfangsadresse der Tabelle gewählt wurde, da dieser Wert nirgends in der verketteten Liste auftreten darf.

Das Einfüge- und das Löschprogramm führen die nötigen Manipulationen der Zeiger durch. Sie verwenden das Flag INDEXED, um anzuzeigen, ob ein Zeiger auf ein Objekt von einer vorhergehenden Eintragung stammt oder vom Inhaltsverzeichnis. Das entsprechende Programm erscheint in Abb. 9.29. Abb. 9.25 zeigt die Datenstruktur.

Ursprüngliche Tabelle

```
DM400
0400  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0410  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0420  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0430  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0440  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0450  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0460  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0470  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

Liste der Objekte und ihrer Adressen im Speicher

```
-DM300
0300  53 4F 4F 31 31 31 31 31 31 31 31 31 31 00 00 00  SOO1111111111...
0310  44 41 44 32 32 32 32 32 32 32 32 32 32 00 00 00  DAD2222222222...
0320  4D 4F 4D 33 33 33 33 33 33 33 33 33 33 00 00 00  MOM3333333333...
0330  55 4F 43 34 34 34 34 34 34 34 34 34 34 00 00 00  UOC4444444444...
0340  41 4E 54 35 35 35 35 35 35 35 35 35 35 00 00 00  ANT5555555555...
0350  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0360  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0370  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

```
-SY
Y=0000 320
-G263/266

F=0266 0266
```

Starte 'INSERT'

Tabelle nach dem Einfügen

```
DM400
0400  4D 4F 4D 33 33 33 33 33 33 33 33 33 33 00 00 00  MOM3333333333...
0410  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0420  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0430  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0440  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0450  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0460  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0470  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

```
-SY
Y=0320 310
-G263/266

F=0266 0266
```

Starte 'INSERT' mit einem anderen Objekt

Liste der Tabelle nach dem Einfügen. Beachten Sie, daß die Tabelle alphabetisch gehalten wird.

```
-DM400
0400  44 41 44 32 32 32 32 32 32 32 32 32 32 4D 4F 4D  DAD2222222222MOM
0410  33 33 33 33 33 33 33 33 33 33 00 00 00 00 00 00  3333333333......
0420  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0430  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0440  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0450  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0460  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0470  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

. . . (weitere Einfügungen) . . .

Abb. 9.24: Alphabetische Liste – Ein Lauf als Beispiel

Tabelle, nachdem alle Objekte eingefügt wurden

```
-DM400
0400  41 4E 54 35 35 35 35 35-35 35 35 35 35 44 41 44  ANT5555555555DAD
0410  32 32 32 32 32 32 32 32-32 32 4D 4F 4D 33 33 33  2222222222MOM333
0420  33 33 33 33 33 33 33 53-4F 4E 31 31 31 31 31 31  3333333SON111111
0430  31 31 31 31 55 4E 43 34-34 34 34 34 34 34 34 34  1111UNC444444444
0440  34 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  4...............
0450  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0460  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0470  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
```

```
-SY
Y=0340 300
-G260/263
P=0263 0263'
```

Starte 'SEARCH' nach 'SON' (bei der Adresse 0300)

Gefunden

```
-DR
 Z  N   A=4E BC=0401 DE=000D HL=0427 S=0100 P=0263 0263' CALL 01D0
        A'=00 B'=0000 D'=0000 H'=0000 X=0427 Y=0300 I=00        (01D0')
```

Adresse von Objekt in der Tabelle

(Überprüfen Sie in der Tabelle, daß dies 'SON' ist)

```
-G266/269
P=0269 0269'
```

Starte 'DELETE' mit 'SON'

Tabelle nach dem Löschen. Beachten Sie, daß UNC nach oben verschoben wurde. Die letzte Eintragung UNC darf nicht beachtet werden.

```
-DM400
0400  41 4E 54 35 35 35 35 35-35 35 35 35 35 44 41 44  ANT5555555555DAD
0410  32 32 32 32 32 32 32 32-32 32 4D 4F 4D 33 33 33  2222222222MOM333
0420  33 33 33 33 33 33 33 55-4E 43 34 34 34 34 34 34  3333333UNC444444
0430  34 34 34 34 55 4E 43 34-34 34 34 34 34 34 34 34  4444UNC444444444
0440  34 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  4...............
0450  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0460  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0470  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
```

```
-G260/263
P=0263 0263'
```

Ein weiterer Versuch von 'SEARCH' (mit 'SON')

Nicht gefunden

```
-DR
S   N   A=FE BC=0401 DE=FF0D HL=0427 S=0100 P=0263 0263' CALL 01D0
        A'=00 B'=0000 D'=0000 H'=0000 X=0427 Y=0300 I=00        (01D0')
-G263/266
P=0266 0266'
```

Füge Objekt wieder ein ('SON')

Momentane Tabelle. Vergleichen Sie sie mit der Tabelle vor Ausführung von DELETE.

```
-DM400
0400  41 4E 54 35 35 35 35 35-35 35 35 35 35 44 41 44  ANT5555555555DAD
0410  32 32 32 32 32 32 32 32-32 32 4D 4F 4D 33 33 33  2222222222MOM333
0420  33 33 33 33 33 33 33 53-4F 4E 31 31 31 31 31 31  3333333SON111111
0430  31 31 31 31 55 4E 43 34-34 34 34 34 34 34 34 34  1111UNC444444444
0440  34 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  4...............
0450  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0460  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0470  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
```

Zeigt, daß der Befehl ausgeführt wurde.

```
-DR
        A=05 BC=FFFF DE=0434 HL=030D S=0100 P=0266 0266' CALL 0221
        A'=00 B'=0000 D'=0000 H'=0000 X=0427 Y=0300 I=00        (0221')
```

Abb.9.24: Alphabetische Liste – ein Lauf als Beispiel (Fortsetzung)

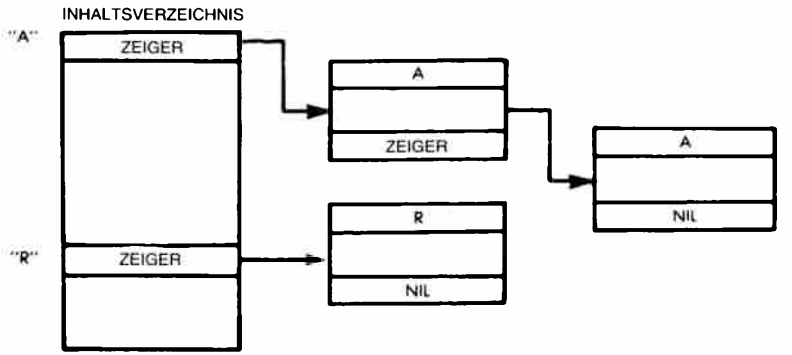
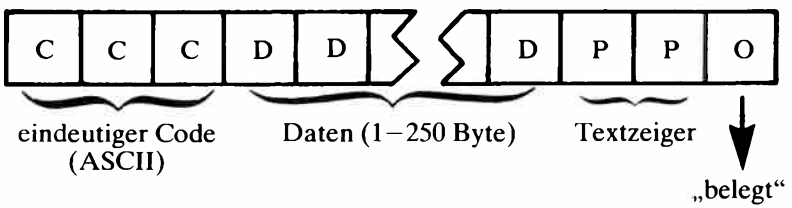


Abb. 9.25: Struktur der verketten Liste

Eine Anwendung für diese Struktur wäre ein computerisiertes Adreßbuch, in dem jede Person durch einen eindeutigen Kode aus drei Buchstaben (vielleicht die Initialien) gekennzeichnet wird und das Datenfeld Adresse und Telefonnummer enthält (bis zu 250 Zeichen). Wir wollen die Struktur in Abb. 9.23 genauer untersuchen. Das Format einer Eintragung ist:



Die üblichen Festlegungen sind:

ENTLEN gesamte Länge (in Byte)
TABASE Adresse des Listenanfangs

Vor Start des Programms muß die Adresse von OBJECT ins Register IY eingetragen werden. REFBASE zeigt hier auf die Anfangsadresse des Inhaltsverzeichnisses.

Jede Zweibyte-Adresse im Inhaltsverzeichnis zeigt auf das erste Auftreten des entsprechenden Buchstabens in der Liste. So bildet jede Gruppe von Eintragungen mit dem gleichen Anfangsbuchstaben der Marke tatsächlich eine getrennte Liste innerhalb der Struktur. Dies vereinfacht die Suche und ist analog zu einem Adreßbuch. Beachten Sie, daß beim Einfügen und Löschen keine Daten bewegt werden. Wie in jeder verketteten Listenstruktur werden nur Zeiger verändert.

Wird keine Eintragung mit einem bestimmten Anfangsbuchstaben gefunden, oder gibt es auf eine Eintragung keine alphabetisch folgende, dann zeigen die entsprechenden Zeiger auf den Tabellenanfang (= „Null"). Dort steht vereinbarungsgemäß ein Wert, der nicht zwischen „A" und „Z" liegt. Er bildet die Marke „Ende der Tabelle" (EOT). Hier wurde angenommen, daß die EOT-Marke den gleichen Speicherbereich einnimmt, wie eine normale Eintragung. Wenn nötig, genügt natürlich auch ein einzelnes Byte. Die Buchstaben sollen alphabetische Zeichen im ASCII-CODE sein. Wollte man dies ändern, müßte man die Konstante in der Routine PRETAB verändern.

Die Markierung Ende der Tabelle ist auf den Wert am Tabellenanfang („Null") gesetzt.

Vereinbarungsgemäß werden die „Nullzeiger", die am Ende eines Strings stehen oder im Inhaltsverzeichnis an einer Stelle, die auf keinen String zeigt, auf den Wert am Tabellenanfang gesetzt, um eine einheitliche Identifikation zu haben. Man könnte auch eine andere Vereinbarung verwenden. Eine andere Markierung für EOT könnte teilweise etwas Platz sparen, da keine Nulleintragungen für leere Stellen vorhanden sein müßten.

Einfügen und Löschen wird auf die übliche Art und Weise durchgeführt (siehe den 1. Teil dieses Kapitels), indem nur die entsprechenden Zeiger geändert werden. Das Flag INDEXED dient dazu anzuzeigen, ob der Zeiger auf das Objekt im Inhaltsverzeichnis zeigt oder auf ein anderes Element.

Suchen

Das Programm SEARCH belegt den Speicherbereich 0100 bis 0155 und es benutzt das Unterprogramm PRETAB bei der Adresse 01D2.

Das Suchprinzip ist einfach.

1 – Lies die Eintragung im Inhaltsverzeichnis, die dem Buchstaben in der ersten Stelle der Marke von OBJECT entspricht.

2 – Lies den Zeiger. Greife auf das Element zu. Wenn Null, dann existiert kein Element.

3 – Falls nicht Null, dann vergleiche das Element mit OBJECT. Ergibt sich Gleichheit, dann war die Suche erfolgreich. Falls nicht, dann lies den Zeiger auf die nächste Eintragung in der Liste.

4 – Gehe zu 2 zurück.

In Abbildung 9.26 ist ein Beispiel gezeigt.

Einfügen

Das Einfügen ist im wesentlichen ein Suchen, dem eine Eintragung folgt, sobald eine „Null" gefunden wurde.

Für die neue Eintragung wird ein Speicherblock hinter der Markierung EOT gesucht, dessen Markierung „verfügbar" angibt.

Das Programm in Abb.9.29 heißt „NEW“ und belegt den Bereich 0156 bis 01A3. Abbildung 9.27 zeigt ein Beispiel.

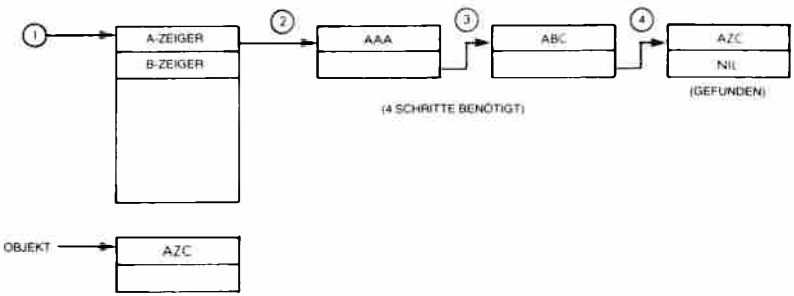


Abb. 9.26: Verkettete Liste – eine Suche

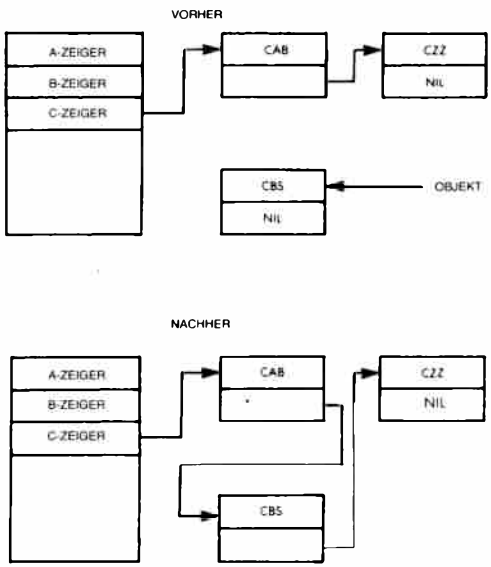


Abb. 9.27: Verkettete Liste: Beispiel für das Einfügen

Löschen

Das Element wird gelöscht, indem die Markierung auf „verfügbar" geändert wird, und der Zeiger im Inhaltsverzeichnis oder im vorhergehenden Element, der auf dieses Element zeigt, geändert wird.

Das Programm heißt „DELETE", und es belegt den Adreßbereich 01A4 bis 01D1.

Abbildung 9.28 zeigt ein Beispiel für das Löschen.

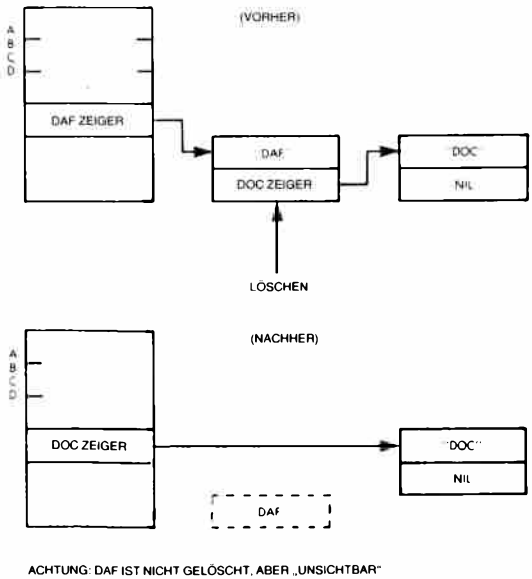


Abb. 9.28: Beispiel für das Löschen (Verkettete Liste)

```
0000'                           ORG     0100H
        (01E7)  INDEXED         DL      INDEX
        (01E8)  TABASE          DL      INDEX1
        (01EA)  REBASE          DL      INDEX3
        (01EC)  ENTLEN          DL      INDEX5
                ;
0100    3E00    SEARCH          LD      A,0             ;INITIALIZE FLAGS
0102    47                      LD      B,A
0103    3C                      INC     A
0104    32E701                  LD      (INDEXED),A
0107    CD0201                  CALL    FRETAB          ;GET ADDR OF INDEX POINTER
010A    1A                      LD      A,(DE)          ;MOVE POINTER CONTENTS TO HL
010B    6F                      LD      L,A
010C    13                      INC     DE
010D    1A                      LD      A,(DE)
010E    67                      LD      H,A
010F    E5                      PUSH    HL
0110    DDE1                    POP     IX
0112    DD7E00  COMPARE         LD      A,(IX+0)        ;LOOK AT FIRST LETTER OF ENTRY
0115    FE7C                    CP      7CH             ;SEE IF IS EOT MARKER
0117    D25501                  JP      NC,NOTFOUND
011A    DD7E00                  LD      A,(IX+0)        ;COMPARE FIRST LETTERS
011D    FDBE00                  CP      (IY+0)
0120    DA3F01                  JP      C,NOGOOD
0123    C25501                  JP      NZ,NOTFOUND
0126    DD7E01                  LD      A,(IX+1)        ;COMPARE 2ND LETTERS
0129    FDBE01                  CP      (IY+1)
012C    DA3F01                  JP      C,NOGOOD
012F    C25501                  JP      NZ,NOTFOUND
0132    DD7E02                  LD      A,(IX+2)        ;COMPARE 3RD LETTERS
0135    FDBE02                  CP      (IY+2)
0138    CA5301                  JP      Z,FOUND
013B    D25501                  JP      NC,NOTFOUND
013E    DDE5    NOGOOD          PUSH    IX
0140    D1                      POP     DE
0141    2AEC01                  LD      HL,(ENTLEN)     ;JUMP TO POINTER OF ENTRY
0144    19                      ADD     HL,DE
0145    4E                      LD      C,(HL)          ;PUT POINTER VALUE IN BC
0146    23                      INC     HL
0147    46                      LD      B,(HL)
0148    C5                      PUSH    BC              ;LOAD IX WITH POINTER
0149    DDE1                    POP     IX
014B    3E00                    LD      A,0
014D    32E701                  LD      (INDEXED),A     ;RESET FLAG
0150    C31201                  JP      COMPARE
0153    06FF    FOUND           LD      B,0FFH
0155    C9      NOTFOUND        RET
                ;
                ;
                ;
0156    CD0001  NEW             CALL    SEARCH          ;SEE WHERE OBJECT SHOULD GO
0159    04                      INC     B
015A    CAA301                  JP      Z,OUT
015D    D5                      PUSH    DE              ;STORE ADDR. OF PREVIOUS ENTRY
015E    2AE801                  LD      HL,(TABASE)     ;FIND SPACE IN TABLE FOR NEW
0161    EB      NEXTONE         EX      DE,HL           ;MOVE TO END OF NEXT ENTRY
0162    2AEC01                  LD      HL,(ENTLEN)
0165    23                      INC     HL              ;ADD 3 FOR REAL LENGTH OF ENTRY
0166    23                      INC     HL
0167    23                      INC     HL
0168    19                      ADD     HL,DE
0169    7E                      LD      A,(HL)
016A    3D                      DEC     A
016B    CA6101                  JP      Z,NEXTONE       ;IF SOMETHING IS THERE, TRY AGAIN
016E    13                      INC     DE
016F    D5                      PUSH    DE              ;SAVE POSITION OF EMPTY SPACE
0170    FDE5                    PUSH    IY              ;MOVE IY TO HL
0172    E1                      POP     HL
0173    ED4BEC01                LD      BC,(ENTLEN)     ;MOVE OBJECT INTO TABLE
0177    EDB0                    LDIR
0179    DDE5                    PUSH    IX              ;PUT ADDR OF ENTRY AFTER OBJECT
017B    E1                      POP     HL              ;..AT POINTER POSITION
017C    EB                      EX      DE,HL
017D    73                      LD      (HL),E
017E    23                      INC     HL
017F    72                      LD      (HL),D
0180    23                      INC     HL
0181    3601                    LD      (HL),1          ;SET OCCUPANCY MARKER
```

Abb. 9.29: Verekettete Liste – die Programme

```
0183  E1              POP   HL              ;GET ADDR OF WHERE THIS SPACE IS
0184  3AE701          LD    A,(INDEXED)     ;SEE WHAT PREVIOUS POINTERS MUST
0187  3D              DEC   A               ;..BE SET
0188  CA9801          JP    Z,SETINX
018B  E3              EX    (SP),HL         ;GET ADDR OF ENTRY PREVIOUS TO
018C  ED5BEC01        LD    DE,(ENTLEN)     ;..OBJECT & MOVE TO POINTER AREA
0190  19              ADD   HL,DE
0191  D1              POP   DE              ;RETRIEVE ADDR OF OBJECT
0192  73              LD    (HL),E          ;PUT IT AT POINTER POSITION
0193  23              INC   HL
0194  72              LD    (HL),D
0195  C3A001          JP    FINISH
0198  C1        SETINX: POP   BC              ;CLEAR OUT STACK
0199  CDD201          CALL  PRETAB          ;GET INDEX ADDRESS
019C  EB              EX    DE,HL           ;LOAD HL INTO IT
019D  73              LD    (HL),E
019E  23              INC   HL
019F  72              LD    (HL),D
01A0  01FFFF  FINISH: LD    BC,0FFFFH       ;SHOW THAT IT WAS DONE
01A3  C9        OUT:  RET
                      ;
                      ;
                      ;
01A4  CD0001    DELETE: CALL  SEARCH          ;GET ADDRESS OF OBJECT
01A7  04              INC   B               ;SEE IF IT IS THERE
01A8  C2D101          JP    NZ,OUTF
01AB  DDE5            PUSH  IX              ;SET HL TO POINTER AREA OF OBJECT
01AD  E1              POP   HL
01AE  ED4BEC01        LD    BC,(ENTLEN)
01B2  09              ADD   HL,BC
01B3  4E              LD    C,(HL)          ;RETRIEVE POINTER
01B4  23              INC   HL
01B5  46              LD    B,(HL)
01B6  23              INC   HL
01B7  3600            LD    (HL),0          ;REMOVE OCCUPANCY MARKER
01B9  3AE701          LD    A,(INDEXED)     ;SEE IF INDEX NEEDS CHANGING
01BC  3D              DEC   A
01BD  C2C701          JP    NZ,CHANGEM
01C0  CDD201          CALL  PRETAB          ;YES,PUT ADDR INTO HL
01C3  EB              EX    DE,HL
01C4  C3CB01          JP    MOVIN
01C7  2AEC01    CHANGEM: LD   HL,(ENTLEN)     ;SET HL TO POINTER OF PREVIOUS
01CA  19              ADD   HL,DE
01CB  71        MOVIN:  LD    (HL),C          ;PUT ADDR OF NEXT INTO WHATEVER
01CC  23              INC   HL                ;..(EITHER INDEX OR ENTRY)
01CD  70              LD    (HL),B
01CE  01FFFF          LD    BC,0FFFFH
01D1  C9        OUTF: RET
                      ;
                      ;
                      ;
01D2  E5        PRETAB: PUSH  HL
01D3  FD7E00          LD    A,(IY+0)        ;GET FIRST LETTER OF OBJECT
01D6  3D              DEC   A               ;REMOVE ASCII LEADER
01D7  D640            SUB   40H
01D9  CB27            SLA   A               ;MULTIPLY BY 2
01DB  2AEA01          LD    HL,(REFBASE)
01DE  85              ADD   L
01DF  6F              LD    L,A
01E0  D2E401          JP    NC,FIXUP
01E3  24              INC   H
01E4  EB        FIXUP:  EX    DE,HL
01E5  E1              POP   HL
01E6  C9              RET

01E7  (0000)    ENDER   END


SYMBOL TABLE

CHANGE  01C7  COMPAR  0112  DELETE  01A4  ENDER   01E7  ENTLEN  01EC
FINISH  01A0  FIXUP   01E4  FOUND   0153  INDEXE  01E7  MOVIN   01CB
NEW     0156  NEXTON  0161  NOGOOD  013E  NOTFOU  0155  OUT     01A3
OUTF    01D1  PRETAB  01D2  REFBAS  01EA  SEARCH  0100  SETINX  0198
TABASE  01EB
```

Abb. 9.29: Verkettete Liste – die Programme (Fortsetzung)

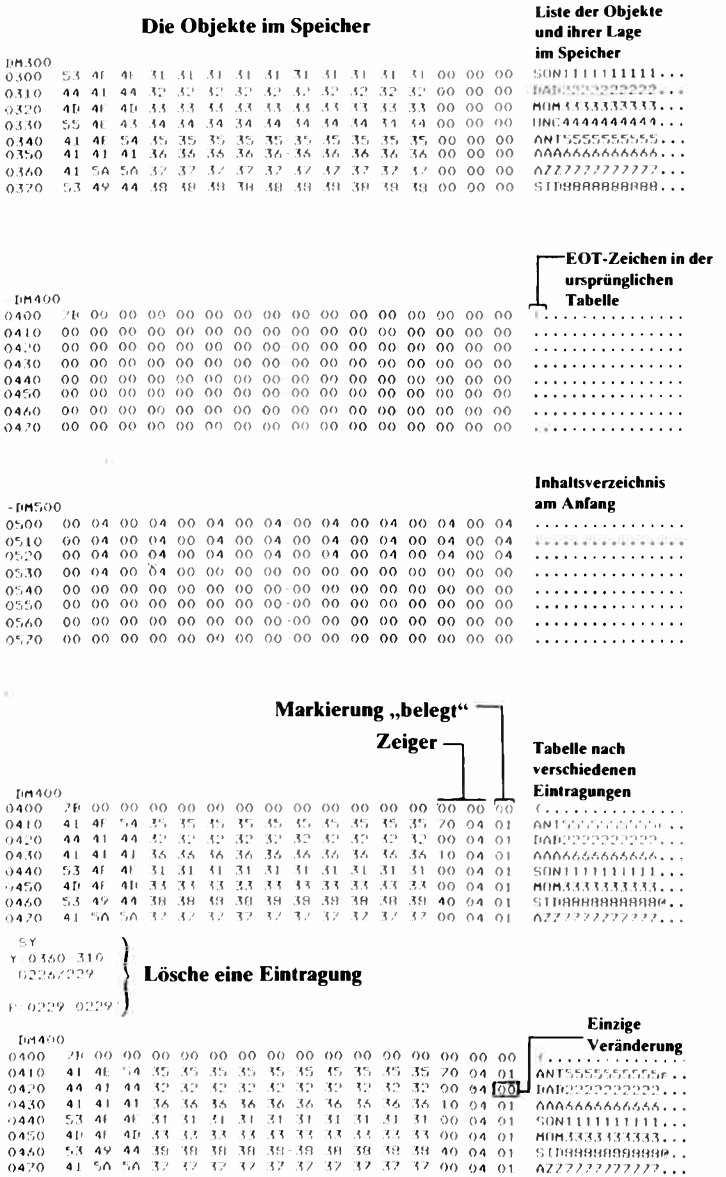


Abb. 9.30: Verkettete Liste – ein Lauf als Beispiel

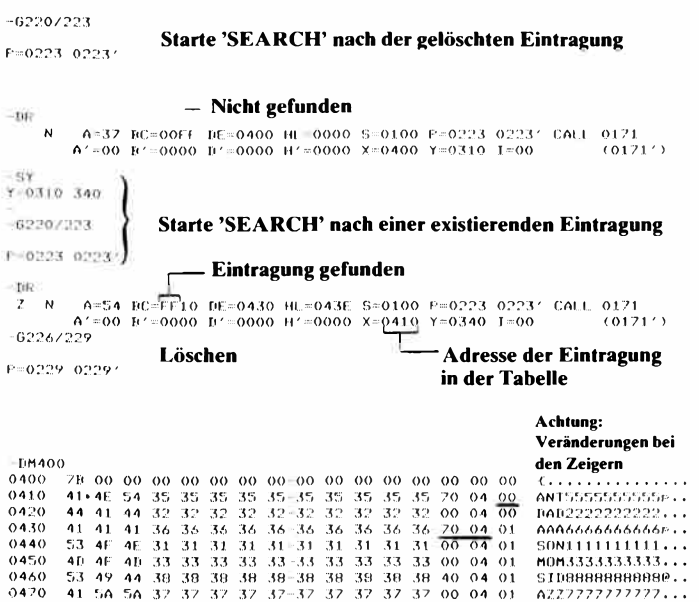


Abb. 9.30: Verkettete Liste – ein Lauf als Beispiel

Zusammenfassung

Der Programmiererneuling braucht sich noch nicht mit den Einzelheiten des Einbaus und der Verwaltung von Datenstrukturen befassen. Effiziente Programmierung von nicht-trivialen Algorithmen setzt jedoch ein gutes Verständnis von Datenstrukturen voraus. Die Beispiele in diesem Kapitel sollen dem Leser helfen, eine solches Verständnis zu erlangen und alle die allgemeinen Probleme zu lösen, die mit geeigneten Datenstrukturen verbunden sind.

10

Entwicklung von Programmen

Einführung

Alle Programme, die wir bisher entwickelt haben, wurden ohne jede Unterstützung durch Software oder Hardware von Hand entwickelt. Die einzige Verbesserung gegenüber der direkten binären Kodierung war die Verwendung mnemotechnischer Symbole, nämlich der Assemblersprache. Um Software effizient entwickeln zu können, muß man wissen, welche Hardware- und Softwarehilfsmittel es gibt. Es ist die Aufgabe dieses Kapitels, diese Hilfsmittel vorzustellen und zu bewerten.

Grundlegende Entscheidungen

Es gibt drei prinzipielle Alternativen: Man kann ein Programm binär oder hexadezimal schreiben, man kann es in Assemblersprache oder in einer höheren Programmiersprache schreiben. Wir wollen diese Möglichkeiten genauer betrachten.

Hexadezimale Kodierung

Ein Programm wird normalerweise in Assemblersprache geschrieben. Viele billige Einplatinencomputer haben jedoch keinen Assembler. Der Assembler ist das Programm, das die Abkürzungen, die man im Programm verwendet, automatisch in den benötigten Binärkode übersetzt. Ist kein Assembler verfügbar, dann muß man diese Übersetzung von der Assemblersprache in den Binärkode von Hand ausführen. Die binäre Darstellung ist nicht bequem zu handhaben und führt leicht zu Fehlern, so daß man normalerweise die hexadezimale Darstellung verwendet. In Kapitel 1 wurde gezeigt, daß eine Hexadezimalziffer vier binäre Bits darstellt. Deshalb nimmt man zwei Hexadezimalziffern, um den Inhalt jedes Bytes anzugeben. Eine Tabelle, die die hexadezimalen Äquivalente der Z80-Befehle angibt, steht im Anhang.

Kurz gesagt, der Programmierer muß dann ein Programm von Hand in hexadezimale Darstellung übersetzen, wenn die Hilfsmittel des Benutzers begrenzt sind und kein Assembler zur Verfügung steht. Dies kann man vernünftigerweise nur mit einer kleinen Anzahl von Befehlen tun (vielleicht mit 10 bis 100). Für umfangreichere Programme ist dieses Verfahren langwierig und fehleranfällig, so daß es nicht zu empfehlen ist. Man muß jedoch bei fast allen Einplatinencomputern die Programme hexadezimal eingeben. Um die Kosten gering zu halten, sind diese nicht mit einem Assembler und einer vollen alphanumerischen Tastatur ausgestattet.

Zusammengefaßt ist die hexadezimale Kodierung keine günstige Art, ein Programm in einem Computer einzugeben. Es ist nur eine billige Art. Man muß die Kosten für einen Assembler und eine Tastatur gegen die Arbeit aufwiegen, ein Programm in den Speicher einzugeben. Dies ändert jedoch nicht die Art und Weise, wie das Programm selbst geschrieben wird. Das Programm wird trotzdem in Assemblersprache geschrieben, so daß es der menschliche Programmierer überprüfen und verstehen kann.

Programmierung in Assemblersprache

Programmierung in Assemblersprache betrifft sowohl die Programme, die man in hexadezimaler Form eingibt, als auch die, die in symbolischer Assemblersprache in den Rechner eingegeben werden können. Wir wollen jetzt untersuchen, wie ein Programm direkt in seiner Darstellung in Assemblersprache eingegeben wird. Dazu braucht man ein Assemblerprogramm. Der Assembler liest jeden mnemonischen Befehl des Programms und übersetzt ihn in das entsprechende Bitmuster, wobei er ein bis fünf Byte belegt, wie es durch die Kodierung des Befehls festgelegt ist. Zusätzlich bietet ein guter Assembler eine Anzahl weiterer Hilfen zum Schreiben des Programms. Diese werden in dem Abschnitt über Assembler unten dargestellt. Speziell sind *Direktiven* verfügbar, die den Wert von Symbolen verändern. Man kann symbolisch adressieren und Sprünge zu symbolischen Adressen angeben. Wenn ein Benutzer während der Testphase Befehle entfernt oder hinzufügt, dann muß man nicht das ganze Programm ändern, wenn ein zusätzlicher Befehl zwischen einem Sprungbefehl und der Stelle eingefügt wird, auf die der Sprung zielt, solange man symbolische Marken verwendet. Der Assembler berücksichtigt das automatisch, wenn er bei der Übersetzung die symbolischen Marken in Adressen umrechnet. Zusätzlich erlaubt es der Assembler, Fehler in der symbolischen Darstellung des Programms zu suchen. Wenn man den Inhalt einer Speicherstelle untersuchen und den Assemblerbefehl, den er darstellt, rekonstruieren will, dann kann man einen Disassembler verwenden. Die verschiedenen Softwarehilfsmittel, die auf einem System normalerweise zur Verfügung stehen, werden unten vorgestellt. Wir wollen jetzt die dritte Alternative untersuchen.

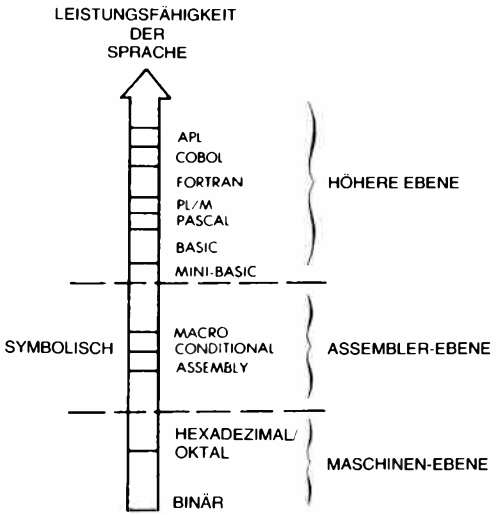


Abb. 10.1: Programmierebenen

Höhere Programmiersprachen

Man kann ein Programm in einer höheren Sprache wie BASIC, APL oder PASCAL schreiben. Die Techniken der Programmierung in diesen verschiedenen Sprachen werden in speziellen Büchern behandelt, und sie werden hier nicht vorgestellt. Wir wollen deshalb diese Art der Programmierung nur kurz skizzieren. Eine höhere Programmiersprache bietet leistungsfähige Anweisungen, die die Programmierung erheblich leichter und schneller machen. Diese Anweisungen müssen dann von einem komplizierten Programm in die endgültige binäre Form umgesetzt werden, die ein Mikroprozessor ausführen kann. Typischerweise wird eine Anweisung in einer höheren Programmiersprache in eine Vielzahl einzelner binärer Befehle übersetzt. Das Programm, das diese Übersetzung durchführt, heißt *Compiler* oder *Interpreter*. Ein Compiler übersetzt alle Anweisungen eines Programms nacheinander in Objektcode. In einer getrennten Phase wird der Code dann ausgeführt. Im Gegensatz dazu interpretiert ein Interpreter einen einzigen Befehl, führt ihn aus, übersetzt dann die nächste Anweisung und führt diese aus. Ein Interpreter bietet den Vorteil einer interaktiven Arbeitsweise, ist aber weniger effizient als ein Compiler. Diese Besonderheiten werden hier nicht weiter studiert. Wir wollen uns der Programmierung eines tatsächlichen Mikroprozessors auf Assemblersprachebene zuwenden.

Software-Unterstützung

Wir wollen hier die hauptsächlichen Softwarehilfsmittel besprechen, die in einem vollständigen System zur Softwareentwicklung vorhanden sind (oder sein sollten). Einige der Begriffe wurden bereits eingeführt. Sie werden hier zusammengefaßt und die weiteren wichtigen Programme werden definiert, bevor wir fortfahren.

Der *Assembler* ist das Programm, das die mnemonische Darstellung von Befehlen in ihr binäres Äquivalent übersetzt. Er übersetzt normalerweise einen symbolischen Befehl in einen binären Befehl (der 1, 2, 3 oder 4 Byte belegen kann). Der resultierende Binärkode wird *Objektkode* genannt. Er kann direkt von dem Mikroprozessor ausgeführt werden. Als Nebeneffekt erzeugt der Assembler eine komplette symbolische Liste des Programms, außerdem eine Äquivalenztabelle für den Programmierer und eine Liste der Symbole in dem Programm. Beispiele werden später in diesem Kapitel vorgestellt.

Zusätzlich listet der Assembler Syntaxfehler, wie Befehl falsch geschrieben oder unzulässig, fehlerhafte Verzweigung, doppelte oder fehlende Marken.

Er erkennt aber keine *logischen* Fehler (das ist Ihr Problem).

Ein *Compiler* ist ein Programm, das Anweisungen aus einer höheren Programmiersprache in eine binäre Form übersetzt.

Ein *Interpreter* ist ein Programm, das ähnlich wie der Compiler auch Anweisungen aus einer höheren Sprache in binären Kode übersetzt, diese aber nicht speichert, sondern sofort ausführt. Tatsächlich wird der binäre Kode oft gar nicht erzeugt, sondern die Anweisungen in der höheren Sprache werden direkt ausgeführt.

Ein *Monitor* ist das Grundprogramm, das unverzichtbar ist, wenn man die Hardwaremöglichkeiten des Systems nutzen will. Er fragt die Eingabegeräte ständig auf Eingaben ab und verwaltet die restlichen Geräte. Beispielsweise muß ein minimaler Monitor für einen Einplatinencomputer, der mit einer Tastatur und einer LED-Anzeige ausgestattet ist, die Tastatur fortlaufend auf Eingaben des Benutzers abfragen und den festgelegten Inhalt auf den Leuchtdioden ausgeben. Zusätzlich muß er eine begrenzte Zahl von Befehlen von der Tastatur verstehen, z. B. START, STOP, CONTINUE (mache weiter), LOAD MEMORY (lade Speicher), EXAMINE MEMORY (überprüfe Speicher). Auf einem großen System bildet der Monitor oft das Überwachungsprogramm, wenn komplexe Dateiverwaltung oder Taskverwaltung verfügbar ist. Die Gesamtheit aller Funktionen heißt dann *Betriebssystem*. Stehen die Dateien auf einer Platte, wird das Betriebssystem als Platten-Betriebssystem oder DOS (Disk Operating System) bezeichnet.

Ein *Editor* ist ein Programm, das die Eingabe und die Korrektur von Texten oder Programmen vereinfacht. Er erlaubt dem Benutzer, Zei-

chen einzugeben, sie anzuhängen oder einzufügen, Zeilen einzufügen und zu löschen und Zeichen oder Zeichenketten zu suchen. Er ist ein wichtiges Hilfsmittel, um Text bequem und effektiv einzugeben.

Ein *Debugger* ist zur Fehlersuche in Programmen nötig. Wenn ein Programm nicht richtig arbeitet, dann ergibt sich oft kein Hinweis, woran das liegt. Deshalb kann der Programmierer in dieses Programm Haltepunkte (Breakpoints) einbauen, um die Ausführung an der angegebenen Adresse zu unterbrechen und an dieser Stelle Inhalte von Registern und Speichern anschauen zu können. Dies ist die hauptsächliche Funktion eines Debuggers. Der Debugger bietet die Möglichkeit, ein Programm zu unterbrechen, die Ausführung fortzusetzen und Inhalte von Registern und Speicherzellen zu überprüfen, anzuzeigen und zu ändern. Ein guter Debugger besitzt oft noch zusätzliche Fähigkeiten, z. B. die, Daten in symbolischer Form, hexadezimal, binär oder in anderen gebräuchlichen Darstellungen zu überprüfen und in diesem Format einzugeben.

Ein *Loader* oder *Linking Loader* plaziert verschiedene Blöcke von Objektkode an angegebene Stellen im Speicher und paßt ihre symbolischen Zeiger so an, daß sie aufeinander zugreifen können. Er wird dazu verwendet, Programme oder Blöcke in verschiedene Speicherbereiche zu legen.

Ein *Simulator* hat die Aufgabe, die Funktion eines Bausteins, meist des Mikroprozessors, zu simulieren, wenn man ein Programm auf einem simulierten Prozessor entwickelt, bevor man es auf die endgültige Karte überträgt.Mit diesem Werkzeug wird es möglich, Programme zu unterbrechen, zu ändern und im Speicher zu halten. Die Nachteile eines Simulators sind:

1 – Normalerweise simuliert er nur den Prozessor, keine Ein-/Ausgabegeräte.

2 – Die Ausführung ist langsam und man arbeitet in einer simulierten Zeitskala. Es ist deshalb nicht möglich, Bausteine unter Echtzeitbedingungen zu testen. Synchronisationsprobleme können aber auch dann noch auftreten, wenn sich die Logik eines Programms schon als fehlerfrei erwiesen hat.

Ein *Emulator* ist im wesentlichen ein Simulator unter Echtzeitbedingungen. Er verwendet einen Prozessor, um einen anderen zu simulieren, und er simuliert ihn in allen Einzelheiten.

Hilfs- oder Dienstprogramme (Utility Routines) sind im wesentlichen alle die Programme, die bei den meisten Anwendungen nötig sind, und von denen sich der Benutzer wünscht, daß der Hersteller sie mitliefert!

Dies kann Programme für Multiplikation, Division und andere arithmetische Operationen einschließen, Routinen zum Verschieben von Blökken, Tests von Zeichen, Treiber für Ein-/Ausgabegeräte usw.

Wie man schrittweise ein Programm entwickelt

Wir wollen jetzt die typische Vorgehensweise bei der Entwicklung eines Programms auf Assemblersprachebene untersuchen. Wir wollen annehmen, daß alle üblichen Softwarehilfsmittel zur Verfügung stehen, um ihre Nützlichkeit zu demonstrieren. Stehen diese in einem speziellen System nicht zur Verfügung, dann kann man trotzdem Programme entwikkeln, die Entwicklung wird aber umständlicher und die Zeit, die man bei der Fehlersuche verbraucht, wird länger werden.

Normalerweise geht man so vor, daß man zuerst einen Algorithmus entwirft und die Datenstrukturen für das Problem festlegt, das gelöst werden soll. Danach wird ein zusammenhängender Satz von Flußdiagrammen entwickelt, die denAblauf des Programms darstellen. Schließlich werden die Flußdiagramme in die Assemblersprache des entsprechenden Mikroprozessors umgesetzt, dies ist die Phase der Kodierung.

Als nächstes gibt man das Programm in einen Rechner ein. Im nächsten Abschnitt wollen wir die Hardwarehilfsmittel ansprechen, die man in dieser Phase benutzt.

Unter Kontrolle des Editors wird das Programm in den RAM-Speicher des Systems eingegeben. Sobald ein Abschnitt des Programms eingegeben ist, z. B. ein oder mehrere Unterprogramme, wird er getestet.

Zuerst verwendet man den Assembler. Steht der Assembler nicht schon im System zur Verfügung, dann lädt man ihn von einem externen Speicher, z. B. einer Platte. Danach wird das Programm assembliert, d. h. in den Binärkode übersetzt. Dies ergibt ein Objektprogramm, das zur Ausführung bereit ist.

Normalerweise erwartet man nicht, daß ein Programm auf Anhieb richtig arbeitet. Um seine Arbeitsweise zu überprüfen, setzt man an kritischen Stellen Haltepunkte, wo man leicht testen kann, ob die Zwischenergebnsse stimmen. Dafür wird der Debugger verwendet. An ausgewählten Stellen legt man Haltepunkte fest. Dann gibt man einen Befehl „Go“ ein, mit dem das Programm gestartet wird. An jedem Haltepunkt wird das Programm automatisch angehalten. Der Programmierer kann dann überprüfen, ob die Daten bis dahin korrekt sind, indem er die Inhalte von Registern und Speicherzellen untersucht. War alles in Ordnung, macht man bis zum nächsten Haltepunkt weiter. Sobald falsche Daten auftreten, hat man einen Fehler im Programm erkannt. An dieser Stelle nimmt sich der Programmierer normalerweise die Liste des Programms vor, und er untersucht, ob die Kodierung stimmt. Kann dabei kein Fehler gefunden werden, dann mag ein logischer Fehler vorliegen, und man muß die Flußdiagramme zu Rate ziehen. Wir wollen hier annehmen, daß die Flußdiagramme von Hand getestet wurden und in Ordnung sind. Der Fehler dürfte dann in der Kodierung liegen. Es wird deshalb nötig werden, einen Teil des Programms zu ändern. Steht die symbolische Darstellung des Programms noch im Speicher, werden wir einfach wieder in den Editor gehen, die entsprechenden Zeilen ändern und

danach den vorhergehenden Ablauf wiederholen. In manchen Systemen mag der Speicher nicht groß genug sein. so daß man die symbolische Darstellung des Programms auf Diskette oder auf Kassette auslagern muß, bevor man den Objektkode ausführt. In einem solchen Fall muß man natürlich die symbolische Darstellung des Programms von dem Externspeicher neu laden, bevor man den Editor startet.

Das obige Verfahren wird solange wiederholt, bis die Ergebnisse stimmen. Wir wollen betonen, daß Vorbeugen besser ist als Heilen. Ein korrekter Entwurf führt typischerweise zu einem Programm, das sehr bald korrekt läuft, sobald die üblichen Tippfehler oder offensichtliche Fehler bei der Kodierung behoben sind. Ein schlampiger Entwurf kann jedoch zu Programmen führen, die eine extrem lange Zeit zur Fehlersuche brauchen. Kurz gesagt lohnt es sich immer, mehr Zeit in den Entwurf zu stecken, um die Fehlersuche zu verkürzen.

Mit diesem Verfahren ist es möglich, die gesamte Organisation des Programms zu testen, es ist jedoch unmöglich, das Programm unter Echtzeitbedingungen mit Ein-/Ausgabebausteinen zu testen. Müssen Ein-/Ausgabebausteine getestet werden, dann besteht die direkte Lösung darin, daß man das Programm in ein EPROM überträgt, dieses auf der Karte einbaut und schaut, ob es funktioniert.

Es gibt dafür allerdings eine bessere Lösung. Es ist die Verwendung eines *In-Circuit-Emulators*. Ein In-Circuit-Emulator benutzt den Mikroprozessor Z80 (oder einen anderen), um den Z80 unter Echtzeitbedingungen nachzubilden. Der Z80 wird physikalisch annulliert. Der Emulator hat ein Kabel mit einem 40-poligen Stecker, der genau die gleiche Belegung hat, wie der Z80. Diesen Stecker kann man in die Anwendungskarte stecken, die man entwickelt. Die Signale, die der Emulator erzeugt, sind genau die des Z80, höchstens etwas langsamer. Der wesentliche Vorteil ist es, daß das Programm, das getestet werden soll, im RAM-Speicher des Entwicklungssystems bleibt. Es erzeugt genau die Signale, die mit den Ein-/Ausgabegeräten kommunizieren, die man verwenden will. Als Ergebnis ist es möglich, das Programm mit all den Hilfsmitteln des Entwicklungssystems zu entwickeln (Editor, Debugger, symbolische Werkzeuge, Dateisystem), während Ein-/Ausgaben unter Echtzeitbedingungen getestet werden.

Zusätzlich bietet ein guter Emulator weitergehende Möglichkeiten, z. B. *Trace* (eine Spur verfolgen). Ein Trace ist die Aufzeichnung der letzten Befehle oder der Zustände auf den verschiedenen Bussen des Systems vor einem Haltepunkt. Kurz gesagt, liefert ein Trace einen Film der Ereignisse, die vor einem Haltepunkt oder vor einem Fehler abliefen. Es kann auch bei einer festgelegten Adresse oder beim Auftreten einer speziellen Bitkombination ausgelöst werden. Eine solche Möglichkeit ist von großem Wert, da es in der Regel zu spät ist, wenn der Fehler erkannt wird. Die Befehle oder die Daten, die den Fehler ausgelöst haben, traten vor dessen Entdeckung auf. Ist ein Trace vorhanden, dann kann der Benutzer herausfinden, welcher Teil seines Programms den

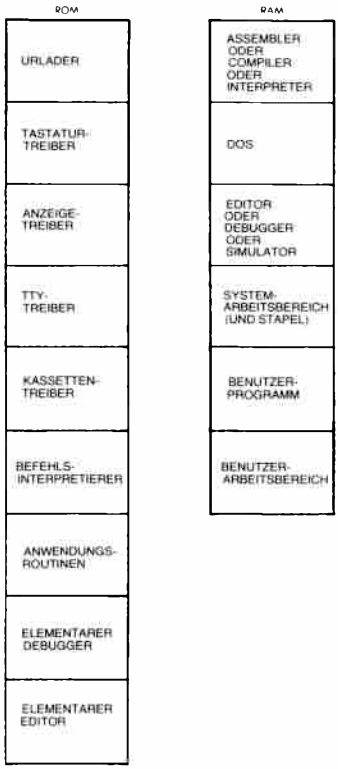


Abb. 10.2: Ein typischer Speicherbelegungsplan

Fehler ausgelöst hat. Ist der Trace nicht lang genug, dann setzt man einfach einen früheren Haltepunkt.

Dies schließt unsere Beschreibung ab, wie man bei der Entwicklung eines Programms normalerweise vorgeht. Wir wollen jetzt die Hardware-Hilfsmittel betrachten, die es zur Programmentwicklung gibt.

Alternativen bei der Hardware

Ein-Platinen-Computer

Ein Ein-Platinen-Computer bietet die billigste Möglichkeit zur Programmentwicklung. Er hat normalerweise eine hexadezimale Tastatur und einige zusätzliche Funktionstasten sowie eine sechsstellige LED-Anzeige, mit der man Adressen und Daten anzeigen kann. Da er nur relativ wenig Speicher hat, ist ein Assembler normalerweise nicht verfügbar. Bestenfalls besitzt er einen kleinen Monitor und außer einigen we-

nigen Befehlen keine Möglichkeiten zum Editieren und zum Debuggen. Deshalb müssen alle Programme hexadezimal eingegeben werden. Sie werden auch in hexadezimaler Form auf den LEDs angezeigt. Ein Ein-Platinen-Computer hat theoretisch die gleiche Hardwareleistung wie jeder andere Computer auch. Er besitzt jedoch nicht alle die nützlichen Hilfsmittel eines größeren Systems und macht die Programmentwicklung viel langwieriger, einfach weil sein Speicherbereich und seine Tastatur eingeschränkt sind. Da es sehr umständlich ist, Programme im hexadezimalen Format zu entwickeln, ist ein Ein-Platinen-Computer zur Aus- und Weiterbildung gut geeignet, da nur Programme beschränkter Länge entwickelt werden und da die begrenzte Länge keine Einschränkung bei der Arbeit bedeutet. Ein-Platinen-Computer sind die wohl billigste Möglichkeit, programmieren zu lernen, indem man es praktiziert. Sie können jedoch nicht zur Entwicklung komplexer Programme verwendet werden, solange man keine zusätzlichen Speicherkarten anschließt und die gebräuchlichen Software-Hilfsmittel verfügbar macht.

Das Entwickungssystem

Ein Entwicklungssystem ist ein Mikrocomputersystem, das einen großen Speicher besitzt (32k, 48k) und auch die benötigten Ein-/Ausgabegeräte, wie Bildschirm, Drucker, Floppy-Disk und üblicherweise einen PROM-Programmierer, vielleicht auch einen In-Circuit-Emulator. Ein Entwicklungssystem wurde speziell dafür entworfen, die Entwicklung von Programmen in industrieller Umgebung zu erleichtern. Es bietet normalerweise alle oder die meisten der Software-Hilfsmittel, die wir im letzten Abschnitt vorgestellt haben. Im Prinzip ist es das ideale Werkzeug zur Entwicklung von Software.

Eingeschränkt ist ein Mikrocomputer-Entwicklungssystem bisweilen in dem Punkt, daß es eventuell nicht in der Lage ist, einen Compiler oder Interpreter zu betreiben. Dies liegt daran, daß ein Compiler typischerweise einen sehr großen Speicherbereich benötigt, oft mehr, als das System enthält. Zur Entwicklung von Programmen auf Assemblersprachebene bietet es jedoch alle nötigen Hilfsmittel. Da Entwicklungssysteme im Vergleich zu Hobbycomputern in relativ kleinen Stückzahlen verkauft werden, sind sie wesentlich teurer.

Mikrocomputer für Hobbyisten

Die Hardware von Hobbycomputern ist natürlich völlig analog zu der eines Entwicklungssystems. Der Hauptunterschied liegt darin, daß ein Hobbycomputer normalerweise nicht mit der hochentwickelten Software ausgerüstet ist, über die ein industrielles Entwicklungssystem verfügt. Beispielsweise bieten viele Mikrocomputer für Hobbyzwecke nur einfache Assembler, einfache Editoren, minimale Dateiverwaltungssysteme und keine Möglichkeit, einen PROM-Programmierer anzuschließen, keinen In-Circuit-Emulator und keinen leistungsfähigen Debugger. Sie bilden deshalb einen Zwischenschritt zwischen den Ein-Platinen-Com-

putern und den vollständigen Entwicklungssystemen. Für einen Anwender, der Programme von mittlerer Komplexität entwickeln will, sind sie wahrscheinlich der beste Kompromiß, da sie bei niedrigen Kosten einen brauchbaren Satz von Software-Hilfsmitteln anbieten, auch wenn sie nur über beschränkten Komfort verfügen.

Time-Sharing Systeme

Man kann sich bei verschiedenen Firmen Terminals mieten, die an Datenübertragungsnetze angeschlossen werden können. Mit einem solchen Terminal erhält man einen Anteil an der Zeitscheibe eines Großrechners und kann von allen Vorteilen einer großen Anlage profitieren. Auf nahezu allen kommerziellen Time-Sharing Systemen sind Cross-Assembler für die meisten Mikroprozessoren verfügbar. Ein Cross-Assembler ist einfach ein Assembler, sagen wir für den Z80, der z. B. auf einer IBM 370 läuft. Formal ist ein Cross-Assembler ein Assembler für den Prozessor X, der auf dem Prozessor Y läuft. Welchen Computer man dazu verwendet, ist ohne Bedeutung. Der Benutzer schreibt die Programme in der Z80-Assemblersprache, und der Cross-Assembler übersetzt sie in die entsprechenden Bitmuster. Der Unterschied ist jedoch, daß das Programm dann nicht sofort ausgeführt werden kann. Es kann mit einem simulierten Prozessor ausgeführt werden, wenn einer zur Verfügung steht und wenn das Programm keine Ein-/Ausgabegeräte anspricht. Dieses Verfahren wird deshalb nur in der Industrie angewendet.

Eigene Rechenanlagen

Verfügt man (in der Firma) über eine eigene Rechenanlage, dann können auch Cross-Assembler zur Programmentwicklung vorhanden sein. Läuft der Rechner im Time-Sharing System, dann entspricht diese Möglichkeit im wesentlichen der oben angegebenen. Bietet der Rechner nur Stapelverarbeitung an (d. h. die eingegebenen Programme werden der Reihe nach abgearbeitet), dann ist die Programmentwicklung sehr unbequem, da die Verarbeitung von Assemblerprogrammen im Stapelbetrieb zu sehr langen Entwicklungszeiten führt.

Anzeige auf der Frontplatte oder nicht?

Die Frontplatte ist ein Hardwarezubehör, das man oft verwendet, um die Fehlersuche in Programmen zu vereinfachen. Traditionell ist sie ein Werkzeug, das den binären Inhalt eines Registers oder eines Speichers bequem anzeigt. Alle Funktionen der Frontplatte kann man jedoch auch von einem Terminal aus durchführen, und mit dem Vordringen der Bildschirmanzeigen werden Möglichkeiten geboten, die der Darstellung binärer Werte auf der Frontplatte vollkommen entsprechen. Der zusätzliche Vorteil einer Bildschirmanzeige ist es, daß man statt der binären Darstellung auch eine hexadezimale, eine symbolische oder eine dezi-

male Darstellung wählen kann (natürlich unter der Voraussetzung, daß die entsprechenden Umwandlungsprogramme zur Verfügung stehen). Der Nachteil eines Bildschirms ist es, daß man mehrere Tasten drücken und nicht nur einen Schalter umlegen muß, um die entsprechende Anzeige zu erhalten. Da jedoch die Kosten für eine Anzeige auf der Frontplatte durchaus entscheidend sind, haben die meisten Hersteller dieses Hilfsmittel für die Fehlersuche verbannt. Der Wert einer Frontplatte wird oft mehr auf der Basis emotionaler Argumente beurteilt, die von den persönlichen Erfahrungen bestimmt sind, als auf der Basis der Zweckmäßigkeit. Sie ist sicher nicht unverzichtbar.

Zusammenfassung der Hardware-Betriebsmittel

Man kann drei hauptsächliche Fälle unterscheiden. Wenn Sie lernen wollen, wie man programmiert, aber nur ein begrenztes Budget haben, dann sei Ihnen der Kauf eines Ein-Platinen-Computers empfohlen. Wenn Sie mit ihm arbeiten, dann können Sie alle die einfachen Programme aus diesem Buch entwickeln und viele weitere. Wenn Sie allerdings Programme aus mehr als einigen hundert Befehlen entwerfen wollen, dann werden Sie an die Grenzen dieser Lösung stoßen.

Wenn Sie industrieller Anwender sind, dann brauchen Sie ein vollständiges Entwicklungssystem. Jede Lösung, die nicht alle Möglichkeiten nutzt, wird eine deutlich längere Entwicklungszeit verursachen. Der Handel ist klar: Hardware-Eigenschaften gegen Programmierzeit. Wenn man nur relativ einfache Programme entwickeln muß, dann kann man natürlich eine billigere Lösung wählen. Wenn jedoch komplexere Programme entwickelt werden müssen, dann kann man kaum irgendwelche Einsparungen an der Hardware rechtfertigen, wenn man ein Entwicklungssystem kauft, da die Kosten für die Programmierung bei jedem Projekt weit dominieren werden.

Für den Hobbyisten bietet ein Hobbycomputer normalerweise ausreichende, wenn auch eingeschränkte Möglichkeiten. Es wird jedoch auch für viele Hobbycomputer gute Entwicklungssoftware zunehmend mehr angeboten. Der Benutzer kann dann sein System um die Komponenten erweitern, die in diesem Kapitel vorgestellt wurden.

Wir wollen jetzt das wichtigste Hilfsmittel genauer untersuchen: den Assembler.

Der Assembler

Das ganze Buch hindurch haben wir die Assemblersprache benutzt, ohne ihre formale Syntax oder ihre Definition vorzustellen. Jetzt ist die Zeit gekommen, diese Definition zu bringen. Eine Assemblersprache wird so entworfen, daß der Benutzer eine bequeme symbolische Darstellung verwenden kann, aber auch so, daß das Assemblerprogramm diese Abkürzungen leicht in ihre binäre Form übersetzen kann.

Assembler-Felder

Wenn wir ein Programm für den Assembler eingegeben haben, dann haben wir gesehen, daß verschiedene Felder verwendet werden. Dies sind:

Das *Markenfeld*, das bei Bedarf eine symbolische Adresse für den laufenden Befehl enthalten kann.

Das *Befehlsfeld*, das den Opcode und irgendwelche Operanden enthält (eventuell kann man ein spezielles Operandenfeld unterscheiden).

Das *Kommentarfeld* ganz rechts, das man wahlweise benutzen kann, und das zur Erklärung eines Programms dient.

Das Programmierformular in Abb. 10.3 zeigt diese Felder.

Wurde der Assembler mit dem Programm gefüttert, dann produziert er eine *Assembler-Liste* davon. Beim Erzeugen der Liste fügt der Assembler drei weitere Felder an, üblicherweise auf der linken Seite. Abb. 10.4 zeigt ein Beispiel. In der dritten Spalte steht die Zeilennummer. Jede Zeile, die der Programmierer eingetippt hat, erhält eine Zeilennummer.

Das nächste Feld rechts ist das aktuelle Adreßfeld, das den hexadezimalen Wert des Befehlszählers enthält, wenn er auf diesen Befehl zeigt.

Rechts davon finden wir die hexadezimale Darstellung des Befehls.

Dies zeigt eine mögliche Anwendung des Assemblers. Auch wenn wir Programme für einen Ein-Platinen-Computer entwerfen, der sie nur hexadezimal akzeptiert, sollten wir die Programme trotzdem auf Assemblerebene schreiben, falls wir Zugang zu einem System haben, auf dem ein Assembler läuft. Wir können das Programm dann auf dem System mit dem Assembler bearbeiten. Der Assembler erzeugt automatisch den richtigen Hexadezimalkode. Dies zeigt den Wert zusätzlicher Software-Hilfsmittel an einem einfachen Beispiel.

Tabellen

Wenn der Assembler das symbolische Programm in die binäre Form übersetzt, führt er im wesentlichen zwei Aufgaben aus:

1 – Er übersetzt die mnemonischen Operationskodes (englisch: Opcodes) in ihre binäre Kodierung.

2 – Er übersetzt die Symbole, die für Konstanten und Adressen verwendet werden, in ihre binäre Darstellung.

Um die Fehlersuche im Programm zu erleichtern, druckt der Assembler am Ende der Liste die Symbole und ihre hexadezimalen Werte. Dies nennt man die Symboltabelle.

Einige Symboltabellen enthalten nicht nur die Symbole und ihre hexadezimalen Werte, sondern auch noch die Nummern der Zeilen, in denen die Symbole auftreten. Damit wird eine zusätzliche Hilfe gegeben.

Fehlermeldungen

Bei der Assemblierung erkennt der Assembler Syntaxfehler und gibt sie in der Liste an. Typische Fehlermeldungen sind: undefinierte Symbole,

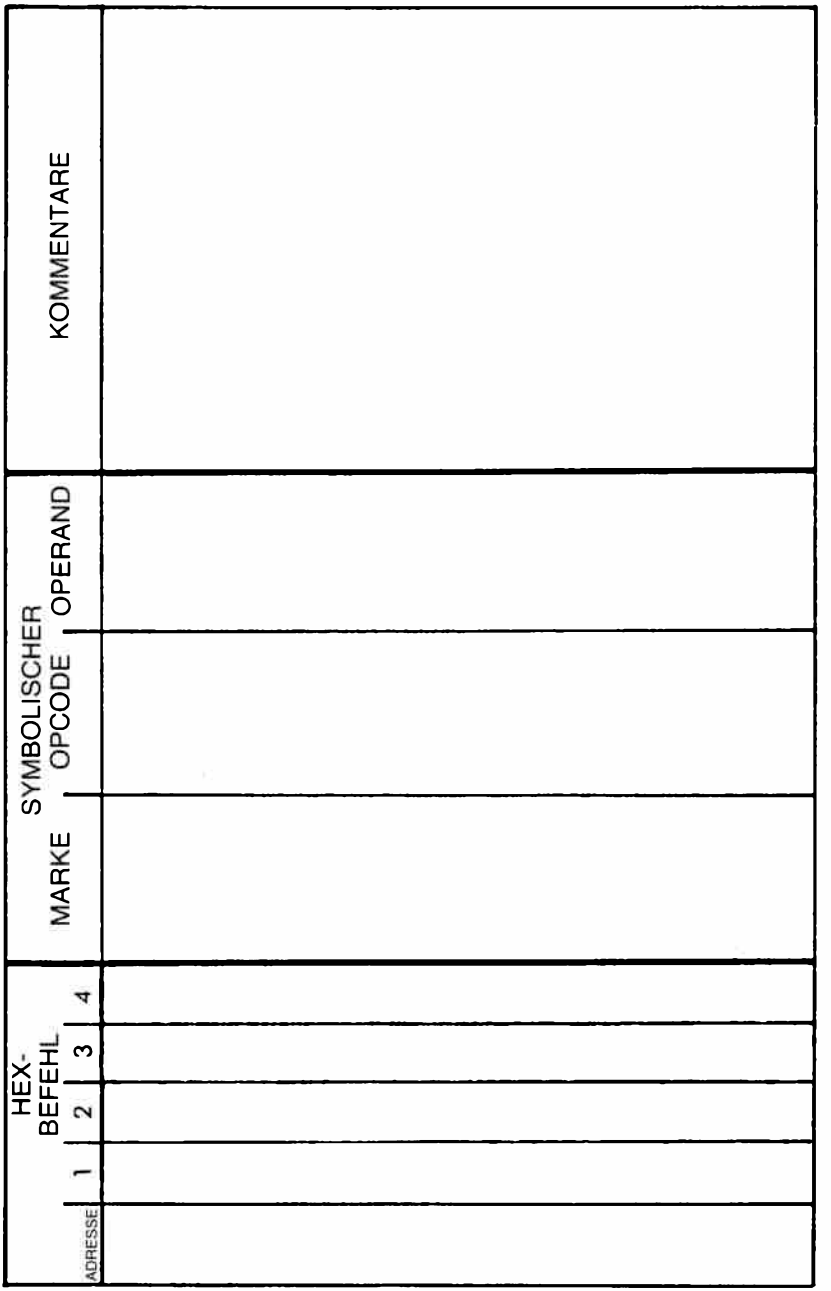


Abb. 10.3: Programmierformular für einen Mikroprozessor

Marke schon definiert, unzulässiger Opcode, unzulässige Adresse, unzulässige Adressierungsart. Viele genauere Fehlermeldungen sind natürlich erwünscht und werden üblicherweise auch geliefert. Dies variiert allerdings von Assembler zu Assembler.

Die Assemblersprache

Opcodes wurden bereits definiert. Wir wollen hier die Symbole definieren, die man als Teil der Assemblersyntax verwenden kann.

Symbole

Symbole dienen dazu, numerische Werte, Daten oder Adressen darzustellen. Symbole dürfen bis zu sechs Zeichen lang sein, und sie müssen mit einem Buchstaben beginnen. Die Zeichen sind auf die Buchstaben des Alphabets und auf die Ziffern beschränkt. Außerdem darf der Benutzer keine Namen verwenden, die mit den Z80-Opcodes, mit den Namen der Register A, B, C, D, E, H, L, AF, BC, DE, HL, IX, IY, SP oder mit einigen weiteren Namen übereinstimmen, die man Pseudoanweisungen des Assemblers nennt. Die Namen dieser „Assemblerdirektiven" sind in dem entsprechenden Abschnitt unten angegeben. Auch die Abkürzungen, die die entsprechenden Flags bezeichnen, sollte man nicht als Symbole verwenden: C, Z, PE, NC, NZ, PO.

Zuordnung eines Werts zu einem Symbol

Marken sind spezielle Symbole, deren Wert der Programmierer nicht zu definieren braucht. Deren Wert wird automatisch vom Assembler festgelegt, immer wenn er eine Marke findet. Der Wert einer Marke ist der Inhalt des Befehlszählers, wenn er auf den Befehl zeigt, vor dem die Marke steht. Will man Marken neue Startwerte zuweisen oder ihnen einen bestimmten Wert geben, dann gibt es dafür spezielle Pseudobefehle.

```
  CROMEMCO CDOS Z80 ASSEMBLER version 02.15                    PAGE 0001

0000                0001        ORG     0100H
      (0200)        0002 MPRAD  DW      0200H
      (0202)        0003 MPDAD  DW      0202H
      (0204)        0004 RESAD  DW      0204H
                    0005 ;
0100  ED4B0002      0006 MP489  LD      BC,(MPRAD)      ;LOAD MULTIPLIER INTO C
0104  0608          0007        LD      B,8             ;8 IS BIT COUNTER
0106  ED5B0202      0008        LD      DE,(MPDAD)      ;LOAD MUTIPLICAND INTO E
010A  1600          0009        LD      D,0             ;CLEAR D
010C  210000        0010        LD      HL,0            ;SET RESULT TO 0
010F  CB39          0011 MULT   SRL     C               ;SHIFT MULTIPLIER BIT INTO CARRY
0111  3001          0012        JR      NC,NOADD        ;TEST CARRY
0113  19            0013        ADD     HL,DE           ;ADD MPD TO RESULT
0114  CB23          0014 NOADD  SLA     E               ;SHIFT MPD LEFT
0116  CB12          0015        RL      D               ;SAVE BIT IN D
0118  05            0016        DEC     B               ;DECREMENT SHIFT COUNTER
0119  C20F01        0017        JP      NZ,MULT         ;DO IT AGAIN IF COUNTER <> 0
011C  220402        0018        LD      (RESAD),HL      ;STORE RESULT
011F  (0000)        0019        END

Errors              0
```

Abb. 10.4: Assemblerliste – ein Beispiel

Andere Symbole, die man für Konstanten oder Speicheradressen verwendet, müssen jedoch vom Programmierer definiert werden, bevor sie benutzt werden.

Um einem Symbol einen Wert zuzuordnen, benutzt man eine spezielle Assemblerdirektive. Eine Direktive ist im wesentlichen ein Befehl an den Assembler, der nicht in eine ausführbare Anweisung übersetzt wird. Beispielsweise wird die Konstante LOG definiert durch:

```
LOG   EQU 3002H
```

Dies weist der Variablen LOG den hexadezimalen Wert 3002 zu. Die Assemblerdirektiven werden später detailliert beschrieben.

Konstanten oder Literals

Konstanten kann man traditionell dezimal, hexadezimal, oktal, binär oder als alphanumerische Zeichenketten darstellen. Damit man die Basis erkennen kann, die zur Darstellung der Zahl verwendet wurde, muß man ein Symbol angeben. Um „0" in den Akkumulator zu laden, schreiben wir einfach:

```
LD   A,0
```

Hinter der Konstanten hätten wir auch ein „D" schreiben können.

Eine hexadezimale Konstante wird durch das Symbol „H" gekennzeichnet. Um den Wert „FF" in den Akkumulator zu laden, schreiben wir:

```
LD   A,FFH
```

Eine oktale Konstante endet auf „O" oder „Q", eine binäre Konstante auf „B".

Um beispielsweise den binären Wert „11111111" in den Akkumulator zu laden, schreiben wir:

```
LD   A,11111111B
```

Auch ASCII-Zeichen im Operandenfeld müssen gekennzeichnet werden. Das ASCII-Symbol wird in Hochkommata eingeschlossen. Um z. B. das Zeichen S in den Akkumulator zu laden, schreiben wir:

```
LD   A,'S'
```

Aufgabe 10.1: Wird mit den beiden folgenden Befehlen der gleiche Wert in den Akkumulator geladen: LD A,'5' und LD A,5H?

Beachten Sie, daß Klammern nach der Zilog-Konvention eine Adresse kennzeichnen. Beispiel:

```
LD   A,(10)
```

gibt an, daß der Akkumulator mit dem Inhalt der Speicherzelle 10 (dezimal) geladen werden soll.

Operatoren

Um das Schreiben symbolischer Programme weiter zu vereinfachen, sollte ein Assembler die Verwendung von Operatoren zulassen. Minimal sollten dies Plus und Minus sein, so daß man beispielsweise schreiben kann:

```
LD  A,(ADRESS)
LD  A,(ADRESS+1)
```

Es ist wichtig zu verstehen, daß der Ausdruck ADRESS+1 vom Assembler ausgerechnet wird, um die tatsächliche Speicheradresse zu erhalten, deren binäres Äquivalent gebildet werden muß. Dieser Ausdruck wird bei der Assemblierung berechnet, nicht bei der Ausführung des Programms.

Zusätzlich können weitere Operatoren möglich sein, z. B. Multiplikation und Division, was die Arbeit mit Tabellen erleichtert. Außerdem kann es weitere spezielle Operatoren geben, wie größer und kleiner, die einen Zweibytewert in sein oberes und sein unteres Byte zerlegen.

In der Regel muß ein Ausdruck ein positives Ergebnis liefern. Normalerweise kann man keine negativen Zahlen verwenden, sondern muß diese hexadezimal angeben.

Um den Inhalt des Befehlszählers in der aktuellen Zeile anzugeben, verwendet man traditionell ein spezielles Symbol: „$". Dieses Symbol sollte man interpretieren als „aktueller Stand" (des Befehlszählers).

Aufgabe 10.2: Was ist der Unterschied zwischen den folgenden Befehlen?

```
LD  A,10101010B
LD  A,(10101010B)
```

Aufgabe 10.3: Welche Wirkung hat der folgende Befehl?

```
JR  NC,$-2
```

Ausdrücke

Die Spezifikation des Z80-Assemblers läßt einen weiteren Bereich von Ausdrücken mit arithmetischen und logischen Befehlen zu. Der Assembler berechnet die Ausdrücke von rechts nach links, unter Beachtung der Prioritäten, die in der Tabelle in Abb. 10.5 angegeben sind. Um eine bestimmte Reihenfolge bei der Berechnung zu erzwingen, kann man Klammern verwenden. Äußere Klammern geben jedoch an, daß der Inhalt der Adresse behandelt werden soll.

Assemblerdirektiven

Direktiven sind spezielle Anweisungeni, die der Programmierer an den Assembler gibt, und die festlegen, daß bestimmte Werte in Symbolen oder im Speicher abgelegt werden, oder die die Druckausgabe des As-

semblers steuern. Die Direktiven, die die Druckausgabe steuern, heißen auch „Kommandos" und werden in einem speziellen Abschnitt beschrieben.

Als Beispiel wollen wir hier die 11 Assemblerdirektiven angeben, die auf dem Zilog-Entwicklungssystem verfügbar sind:

ORG nn

Diese Direktive setzt den Befehlszähler des Assemblers auf den Wert nn. Mit anderen Worten, der erste ausführbare Befehl nach dieser Direktive steht bei der Adresse nn. Mit dieser Direktive können verschiedene Programmteile für verschiedene Speicherbereiche festgelegt werden.

EQU nn

Diese Direktive dient dazu, einem Symbol den Wert nn zuzuweisen.

DEFL nn

Auch diese Direktive weist einem Symbold den Wert nn zu, darf aber innerhalb des Programms mehrmals mit verschiedenen Werten für das gleiche Symbol wiederholt werden, während EQU nur einmal verwendet werden darf.

DEFB n

Diese Direktive weist der Speicherzelle, auf die momentan der Befehlszähler zeigt, den Inhalt n zu.

DEFB 'S'

wiest derSpeicherzelle den ASCII-Wert von „S" zu.

DEFW nn

In die Speicherzelle, auf die momentan der Befehlszähler zeigt, und in die Zelle dahinter wird der Zweibytewert nn eingetragen.

DEFS nn

reserviert einen Speicherblock der Länge nn Byte, beginnend bei der aktuellen Adresse des Befehlszählers.

DEFM 'S'

speichert die angegebene Zeichenkette ab der Speicherstelle, auf die der Befehlszähler zeigt. Die Zeichenkette darf maximal 63 Byte lang sein.

MACRO P0 P1 . . . Pn

dient dazu, eine Marke als Makro zu definieren, und die formale Parameterliste anzugeben. Makros werden weiter unten definiert.

END

kennzeichnet das Programmende.

ENDM

kennzeichnet das Ende eines Makros.

OPERATOR	FUNKTION	PRIORITÄT
+	VORZEICHEN PLUS	1
−	VORZEICHEN MINUS	1
.NOT. or \	LOGISCHES NICHT	1
.RES.	ERGEBNIS	1
**	POTENZ	2
*	MULTIPLIKATION	3
/	DIVISION	3
.MOD.	MODULO	3
.SHR.	LOGISCH RECHTS SCHIEBEN	3
.SHL.	LOGISCH LINKS SCHIEBEN	3
+	ADDITION	4
-	SUBTRAKTION	4
.AND. or &	LOGISCHES UND	5
.OR. or !	LOGISCHES ODER	6
.XOR.	LOGISCHES EXKLUSIVES ODER	6
.EQ. or =	GLEICH	7
.GT. or >	GRÖSSER ALS	7
.LT. or <	KLEINER ALS	7
.UGT.	BETRAGSMÄSSIG GRÖSSER ALS	7
.ULT.	BETRAGSMÄSSIG KLEINER ALS	7

Abb. 10.5: Priorität der Operatoren

Assembler-Kommandos

Kommandos dienen dazu, das Format der Liste zu steuern, die der Assembler erzeugt. Alle Kommandos beginnen mit einem Stern in Spalte Eins. Der Z80-Assembler erkennt sieben Kommandos. Typische Beispiele sind:

```
*EJECT
```

was einen Seitenvorschub veranlaßt und

```
*LIST OFF
```

bewirkt, daß mit diesem Befehl der Listenausdruck beendet wird. Die anderen Kommandos: *HEADING S, *LIST ON, *MACLIST ON, *MACLIST OFF, *INCLUDE FILENAME.

Makros

Ein Makro ist einfach ein Name, den man einer Gruppe von Befehlen gibt. Wird die gleiche Gruppe von Befehlen mehrfach in einem Programm benutzt, dann kann man ein Makro definieren, das sie repräsentiert, anstatt jedesmal die ganze Gruppe zu schreiben. Wir könnten beispielsweise schreiben:

```
SAVREG MACRO
       PUSH AF
       PUSH BC
       PUSH DE
       PUSH HL
       ENDM
```

Statt die obigen Befehle zu schreiben, genügt dann die Angabe des Namens „SAVREG". Immer wenn wir SAVREG schreiben, wird dieser Name durch die entsprechenden fünf Zeilen ersetzt. Ein Assembler, der Makros verarbeiten kann, wird Makro-Assembler genannt. Immer wenn der Makro-Assembler ein SAVREG findet, ersetzt er es durch die entsprechenden Zeilen.

Makro oder Unterprogramm

An dieser Stelle sieht es so aus, als ob ein Makro wie ein Unterprogramm funktioniert. Dies ist aber nicht der Fall. Wenn der Assembler den Objektkode erzeugt, dann ersetzt er jeden Makronamen durch die Befehle, für die er steht. Zur Ausführungszeit erscheint die Gruppe von Befehlen, so oft in dem Programm, wie es der Name des Makros tat.

Im Gegensatz dazu wird ein Unterprogramm nur einmal definiert und gespeichert und kann dann wiederholt aufgerufen werden. Das Programm springt dann zur Adresse des Unterprogramms. Ein Makro ist ein *Hilfsmittel bei der Assemblierung*. Ein Unterprogramm ist ein *Hilfsmittel zur Ausführungszeit*. Die Arbeitsweise ist vollkommen verschieden.

Makroparameter

Jeder Makro kann mit einer Anzahl Parameter versehen werden. Als Beispiel wollen wir den folgenden Makro betrachten:

```
SWAP MACRO M,N,T
     LD    A,M    M nach A
     LD    T,A    A nach T (=M)
     LD    A,N    N nach A
     LD    M,A    A nach M (=N)
     LD    A,T    T nach A
     LD    N,A    A nach N (=T)
     ENDM
```

Dieser Makro bewirkt, daß die Inhalte der Speicherzellen M und N vertauscht werden. Ein Austausch zwischen zwei Registern oder zwei Speicherstellen ist eine Operation, die es beim Z80 nicht gibt. Mit einem Makro kann man sie einbauen. „T" ist in diesem Fall einfach der Name eines Zwischenspeichers, den das Programm benötigt. Als Beispiel wollen wir die Inhalte der Speicherzellen ALPHA und BETA vertauschen. Der Befehl, der dies tut, ist folgender:

```
SWAP (ALPHA),(BETA),(TEMP)
```

In diesem Befehl ist TEMP der Name eines Zwischenspeichers, von dem wir wissen, daß er frei ist, und den der Makro benutzen kann. Der Assembler erzeugt dann folgende Makroerweiterung:

```
LD  A,(ALPHA)
LD  (TEMP),A
LD  A,(BETA)
LD  (ALPHA),A
LD  A,(TEMP)
LD  (BETA),A
```

Die Nützlichkeit eines Makros sollte jetzt klar sein. Für den Programmierer ist es bequem, Pseudobefehle zu verwenden, die durch Makros definiert wurden. Auf diese Art kann der tatsächliche Befehlssatz des Z80 beliebig erweitert werden. Man muß sich allerdings darüber im klaren sein, daß jeder Makro unglücklicherweise in so viele Befehle erweitert wird, wie sie in der Definition benutzt wurden. Ein Makro wird deshalb erheblich langsamer abgearbeitet, als ein einzelner Befehl, allerdings schneller als ein Unterprogramm, da dort noch Unterprogrammaufruf und Rücksprung bearbeitet werden müssen. Wegen der Bequemlichkeit bei der Entwicklung längerer Programme wünscht man sich die Makroeigenschaften für solche Anwendungen.

Zusätzliche Möglichkeiten von Makros

Zur einfachen Makrofähigkeit können noch viele andere Direktiven und syntaktische Möglichkeiten hinzukommen. Makros können *verschachtelt* werden, d. h. Makroaufrufe können innerhalb einer Makrodefinition auftreten. Mit dieser Eigenschaft kann sich ein Makro mit einer verschachtelten Definition selbst modifizieren! Ein erster Aufruf erzeugt eine Erweiterung, während jeder weitere Aufruf aus demselben Makro eine andere Erweiterung erzeugt. Dies ist beim Z80-Assembler zugelassen, verschachtelte Definitionen sind dagegen nicht erlaubt.

Bedingte Assemblierung

Bedingte Assemblierung ist eine andere Möglichkeit, die der Z80-Assembler bietet. Mit der Möglichkeit einer bedingten Assemblierung kann der Programmierer ein Programm für verschiedene Fälle entwerfen und dann die Segmente des Kodes, die für eine spezielle Anwendung gebraucht werden, bedingt assemblieren. Beispielsweise kann ein industrieller Benutzer ein Programm zur Steuerung beliebig vieler Ampeln an einer Straßenkreuzung für verschiedene Steueralgorithmen schreiben. Er bekommt dann von dem lokalen Verkehrsplaner die Angaben, wie viele Ampeln an einer Kreuzung stehen sollen und welcher Steueralgorithmus verwendet werden soll. Der Programmierer setzt dann einfach die entsprechenden Parameter in seinem Programm und assembliert bedingt. Die bedingte Assemblierung führt dann zu einem „Benutzerprogramm", das nur die Routinen enthält, die für die Lösung dieses speziellen Problems benötigt werden.

Die bedingte Assemblierung ist deshalb von besonderem Wert beim Schreiben industrieller Programme für Anwendungen, bei denen es verschiedene Optionen gibt, und bei denen der Programmierer Programmteile abhängig von äußeren Parametern schnell und automatisch assemblieren will.

In der Standardversion des Makro-Assemblers, den Zilog liefert, sind nur zwei bedingte Pseudobefehle eingebaut:

COND NN und ENDC

wobei NN einen Ausdruck darstellt. Der Pseudobefehl „COND NN" veranlaßt die Berechnung des Ausdrucks NN. Ergibt sich der Wert „wahr" (nicht Null), dann werden die Befehle hinter COND assembliert. Ergibt sich jedoch der Wert „falsch", d. h. wird der Wert Null berechnet, dann werden alle folgenden Befehle bis zu demBefehl ENDC nicht assembliert.

ENDC dient dazu, ein COND zu beenden, so daß die nachfolgenden Befehle in jedem Fall assembliert werden. Der Pseudobefehl COND kann nicht verschachtelt werden.

Theoretisch könnte es noch leistungsfähigere Möglichkeiten der bedingten Assemblierung geben, mit den Spezifikationen „IF" und „ELSE". Diese könnten in zukünftigen Versionen des Assemblers verfügbar werden.

Zusammenfassung

Dieses Kapitel hat die Techniken und die Hardware- und Software-Werkzeuge vorgestellt, die man zur Entwicklung eines Programms braucht, zusammen mit den verschiedenen Vorzügen und Alternativen.

Bei der Hardware reichen diese von Ein-Platinen-Computern bis zum kompletten Entwicklungssystem, bei der Software von der einfachen binären Eingabe bis zu den höheren Programmiersprachen.

Die Auswahl müssen Sie treffen, je nach Ihren Zielen und Möglichkeiten.

11

Schluß

Wir haben jetzt alle wichtigen Aspekte der Programmierung abgehandelt, von den Definitionen und Grundkonzepten über die interne Manipulation der Register des Z80, die Verwaltung der Ein-/Ausgabegeräte bis hin zur Beschreibung der Hilfsmittel zur Software-Entwicklung. Was ist der nächste Schritt? Dies kann man aus zwei Blickwinkeln betrachten. Der erste bezieht sich auf die Entwicklung der Technologie, der zweite auf die Weiterentwicklung Ihrer eigenen Kenntnisse und Fähigkeiten. Diese beiden Punkte wollen wir noch ansprechen.

Die technologische Entwicklung

Der Fortschritt bei der Integration von MOS-Strukturen macht es möglich, immer komplexere Chips herzustellen. Die Kosten für den Einbau der Prozessorfunktionen selbst nehmen ständig ab. Das Ergebnis ist, daß viele Ein-/Ausgabechips oder Peripheriesteuerbausteine, die man in einem System verwendet, selbst schon einen einfachen Prozessor enthalten. Dies heißt, daß die LSI-Bausteine in dem System *programmierbar* werden. Daraus entwickelt sich ein interessantes Dilemma der Konzeption. Um die Software-Entwicklung zu vereinfachen und die Zahl der Bauteile im System zu verringern, enthalten die neuen Chips hochentwickelte programmierbare Eigenschaften: Viele programmierte Algorithmen werden jetzt in den Chip eingebaut. Als Ergebnis wird die Entwicklung von Programmen jedoch dadurch erschwert, daß alle diese Ein-/Ausgabegeräte völlig unterschiedlich sind und vom Programmierer genau studiert werden müssen! *Die Programmierung des Systems ist nicht mehr alleine die Programmierung des Mikroprozessors, sondern auch die Programmierung aller anderen Chips, die daran angeschlossen sind.* Die Zeit, die man braucht, um den Umgang mit einem Chip zu lernen, kann durchaus erheblich sein.

Natürlich ist dies nur ein scheinbarer Zwiespalt. Gäbe es diese Bausteine nicht, dann wäre das Interface, das man aufbauen muß, und auch die entsprechenden Programme erheblich komplizierter. Die Schwierig-

keit, die neu auftritt, ist, daß man mehr als nur einen Prozessor programmieren und dazu die unterschiedlichen Eigenschaften der verschiedenen Bausteine eines Systems lernen muß. Es ist jedoch zu hoffen, daß die Techniken und Konzepte, die in diesem Buch vorgestellt wurden, diese Aufgabe verhältnismäßig einfach machen.

Der nächste Schritt

Sie haben jetzt die grundlegenden Techniken gelernt, die man braucht, um einfache Anwendungen auf dem Papier zu lösen. Dies war das Ziel dieses Buchs. Der nächste Schritt ist die Praxis, und dafür gibt es keinen Ersatz. Es ist unmöglich, Programmieren nur auf dem Papier zu lernen. Praktische Übung ist unumgänglich. Sie sollten jetzt in der Lage sein, eigene Programme zu schreiben. Wir hoffen, daß Sie Spaß daran haben.

Denjenigen von Ihnen, die meinen, sie könnten aus einem zusätzlichen Buch Nutzen ziehen, sei das Z80 Applikationsbuch empfohlen. Dieses Buch ist die begleitende Ausgabe zum vorliegenden Buch und es stellt einen weiten Bereich von echten Anwendungen vor, die man mit einem Z80-Mikrocomputer verwirklichen kann.

Anhang A

Tabelle zur Umwandlung von Hexadezimalzahlen

HEX	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	00	000
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	0
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	256	4096
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	512	8192
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	768	12288
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	1024	16384
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	1280	20480
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	1536	24576
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	1792	28672
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	2048	32768
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	2304	36864
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	2560	40960
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	2816	45056
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	3072	49152
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	3328	53248
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	3584	57344
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	3840	61440

5		4		3		2		1		0	
HEX	DEZ	HEX	DEZ	HEX	DEZ	HEX	DEZ	HEX	DEZ	HEX	DEZ
0	0	0	0	0	0	0	0	0	0	0	0
1	1,048,576	1	65,536	1	4,096	1	256	1	16	1	1
2	2,097,152	2	131,072	2	8,192	2	512	2	32	2	2
3	3,145,728	3	196,608	3	12,288	3	768	3	48	3	3
4	4,194,304	4	262,144	4	16,384	4	1,024	4	64	4	4
5	5,242,880	5	327,680	5	20,480	5	1,280	5	80	5	5
6	6,291,456	6	393.216	6	24,576	6	1,536	6	96	6	6
7	7,340,032	7	458,752	7	28,672	7	1,792	7	112	7	7
8	8,388,608	8	524,288	8	32,768	8	2,048	8	128	8	8
9	9,437,184	9	589,824	9	36,864	9	2,304	9	144	9	9
A	10,485,760	A	655,360	A	40,960	A	2,560	A	160	A	10
B	11,534,336	B	720,896	B	45,056	B	2,816	B	176	B	11
C	12,582,912	C	786,432	C	49,152	C	3,072	C	192	C	12
D	13,631,488	D	851,968	D	53,248	D	3,328	D	208	D	13
E	14,680,064	E	917,504	E	57,344	E	3,584	E	224	E	14
F	15,728,640	F	983,040	F	61,440	F	3,840	F	240	F	15

Anhang B

ASCII-Tabelle

HEX	MSD	0	1	2	3	4	5	6	7
LSD	BITS	000	001	010	011	100	101	110	111
0	0000	NUL	DLE	SPACE	0	@	P	–	p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	'	7	G	W	g	w
8	1000	BS	CAN	(	8	H	X	h	x
9	1001	HT	EM	)	9	I	Y	i	y
A	1010	LF	SUB	*	:	J	Z	j	z
B	1011	VT	ESC	+	;	K	[	k	{
C	1100	FF	FS	,	<	L	\	l	--
D	1101	CR	GS	–	=	M	]	m	}
E	1110	SO	RS	.	>	N	^	n	~
F	1111	SI	US	/	?	O	←	o	DEL

Die ASCII-Symbole

NUL – Null
SOH – Start of Heading
STX – Start of Text
ETX – End of Text
EOT – End of Transmission
ENQ – Enquiry
ACK – Acknowledge
BEL – Bell
BS – Backspace
HT – Horizontal Tabulation
LF – Line Feed
VT – Vertical Tabulation
FF – Form Feed
CR – Carriage Return
SO – Shift Out
SI – Shift In
DLE – Data Link Escape
DC – Device Control
NAK – Negative Acknowledge
SYN – Synchronous Idle
ETB – End of Transmission Block
CAN – Cancel
EM – End of Medium
SUB – Substitute
ESC – Escape
FS – File Separator
GS – Group Separator
RS – Record Separator
US – Unit Separator
SP – Space (Blank)
DEL – Delete

Anhang C

Tabelle relativer Sprungdistanzen

Relative Vorwärtssprünge

MSD \ LSD	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127

Relative Rückwärtssprünge

MSD \ LSD	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8	128	127	126	125	124	123	122	121	120	119	118	117	116	115	114	113
9	112	111	110	109	108	107	106	105	104	103	102	101	100	99	98	97
A	96	95	94	93	92	91	90	89	88	87	86	85	84	83	82	81
B	80	79	78	77	76	75	74	73	72	71	70	69	68	67	66	65
C	64	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49
D	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33
E	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17
F	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1

Anhang D

Umwandlung Dezimal nach BCD

DECIMAL	BCD	DEC	BCD	DEC	BCD
0	0000	10	00010000	90	10010000
1	0001	11	00010001	91	10010001
2	0010	12	00010010	92	10010010
3	0011	13	00010011	93	10010011
4	0100	14	00010100	94	10010100
5	0101	15	00010101	95	10010101
6	0110	16	00010110	96	10010110
7	0111	17	00010111	97	10010111
8	1000	18	00011000	98	10011000
9	1001	19	00011001	99	10011001

Anhang E

Der Befehlssatz des Z80

OBJ CODE	QUELLBEFEHL	
8E	ADC	A,(HL)
DD8E05	ADC	A,(IX+d)
FD8E05	ADC	A,(IY+d)
8F	ADC	A,A
88	ADC	A,B
89	ADC	A,C
8A	ADC	A,D
8B	ADC	A,E
8C	ADC	A,H
8D	ADC	A,L
CE20	ADC	A,n
ED4A	ADC	HL,BC
ED5A	ADC	HL,DE
ED6A	ADC	HL,HL
ED7A	ADC	HL,SP
86	ADD	A,(HL)
DD8605	ADD	A,(IX+d)
FD8605	ADD	A,(IY+d)
87	ADD	A,A
80	ADD	A,B
81	ADD	A,C
82	ADD	A,D
83	ADD	A,E
84	ADD	A,H
85	ADD	A,L
C620	ADD	A,n
09	ADD	HL,BC
19	ADD	HL,DE
29	ADD	HL,HL
39	ADD	HL,SP
DD09	ADD	IX,BC
DD19	ADD	IX,DE
DD29	ADD	IX,IX
DD39	ADD	IX,SP
FD09	ADD	IY,BC
FD19	ADD	IY,DE
FD29	ADD	IY,IY
FD39	ADD	IY,SP
A6	AND	(HL)
DDA605	AND	(IX+d)
FDA605	AND	(IY+d)
A7	AND	A
A0	AND	B
A1	AND	C
A2	AND	D
A3	AND	E
A4	AND	H
A5	AND	L

OBJ CODE	QUELLBEFEHL	
E620	AND	n
CB46	BIT	0,(HL)
DDCB0546	BIT	0,(IX+d)
FDCB0546	BIT	0,(IY+d)
CB47	BIT	0,A
CB40	BIT	0,B
CB41	BIT	0,C
CB42	BIT	0,D
CB43	BIT	0,E
CB44	BIT	0,H
CB45	BIT	0,L
CB4E	BIT	1,(HL)
DDCB054E	BIT	1,(IX+d)
FDCB054E	BIT	1,(IY+d)
CB4F	BIT	1,A
CB48	BIT	1,B
CB49	BIT	1,C
CB4A	BIT	1,D
CB4B	BIT	1,E
CB4C	BIT	1,H
CB4D	BIT	1,L
CB56	BIT	2,(HL)
DDCB0556	BIT	2,(IX+d)
FDCB0556	BIT	2,(IY+d)
CB57	BIT	2,A
CB50	BIT	2,B
CB51	BIT	2,C
CB52	BIT	2,D
CB53	BIT	2,E
CB54	BIT	2,H
CB55	BIT	2,L
CB5E	BIT	3,(HL)
DDCB055E	BIT	3,(IX+d)
FDCB055E	BIT	3,(IY+d)
CB5F	BIT	3,A
CB58	BIT	3,B
CB59	BIT	3,C
CB5A	BIT	3,D
CB5B	BIT	3,E
CB5C	BIT	3,H
CB5D	BIT	3,L
CB66	BIT	4,(HL)
DDCB0566	BIT	4,(IX+d)
FDCB0566	BIT	4,(IY+d)
CB67	BIT	4,A
CB60	BIT	4,B
CB61	BIT	4,C
CB62	BIT	4,D

OBJ CODE	QUELLBEFEHL	
CB63	BIT	4,E
CB64	BIT	4,H
CB65	BIT	4,L
CB6E	BIT	5,(HL)
DDCB056E	BIT	5,(IX+d)
FDCB056E	BIT	5,(IY+d)
CB6F	BIT	5,A
CB68	BIT	5,B
CB69	BIT	5,C
CB6A	BIT	5,D
CB6B	BIT	5,E
CB6C	BIT	5,H
CB6D	BIT	5,L
CB76	BIT	6,(HL)
DDCB0576	BIT	6,(IX+d)
FDCB0576	BIT	6,(IY+d)
CB77	BIT	6,A
CB70	BIT	6,B
CB71	BIT	6,C
CB72	BIT	6,D
CB73	BIT	6,E
CB74	BIT	6,H
CB75	BIT	6,L
CB7E	BIT	7,(HL)
DDCB057E	BIT	7,(IX+d)
FDCB057E	BIT	7,(IY+d)
CB7F	BIT	7,A
CB78	BIT	7,B
CB79	BIT	7,C
CB7A	BIT	7,D
CB7B	BIT	7,E
CB7C	BIT	7,H
CB7D	BIT	7,L
DC8405	CALL	C,nn
FC8405	CALL	M,nn
D48405	CALL	NC,nn
C48405	CALL	NZ,nn
F48405	CALL	P,nn
EC8405	CALL	PE,nn
E48405	CALL	PO,nn
CC8405	CALL	Z,nn
CD8405	CALL	nn
3F	CCF	
BE	CP	(HL)
DDBE05	CP	(IX+d)
FDBE05	CP	(IY+d)
BF	CP	A
B8	CP	B
B9	CP	C
BA	CP	D
BB	CP	E
BC	CP	H
BD	CP	L
FE20	CP	n
EDA9	CPD	
EDB9	CPDR	

OBJ CODE	QUELLBEFEHL	
EDB1	CPIR	
EDA1	CPI	
2F	CPL	
27	DAA	
35	DEC	(HL)
DD3505	DEC	(IX+d)
FD3505	DEC	(IY+d)
3D	DEC	A
05	DEC	B
0B	DEC	BC
0D	DEC	C
15	DEC	D
1B	DEC	DE
1D	DEC	E
25	DEC	H
2B	DEC	HL
DD2B	DEC	IX
FD2B	DEC	IY
2D	DEC	L
3B	DEC	SP
F3	DI	
102E	DJNZ	e
FB	EI	
E3	EX	(SP),HL
DDE3	EX	(SP),IX
FDE3	EX	(SP),IY
08	EX	AF,AF'
EB	EX	DE,HL
D9	EXX	
76	HALT	
ED46	IM	0
ED56	IM	1
ED5E	IM	2
ED78	IN	A,(C)
ED40	IN	B,(C)
ED48	IN	C,(C)
ED50	IN	D,(C)
ED58	IN	E,(C)
ED60	IN	H,(C)
ED68	IN	L,(C)
34	INC	(HL)
DD3405	INC	(IX+d)
FD3405	INC	(IY+d)
3C	INC	A
04	INC	B
03	INC	BC
0C	INC	C
14	INC	D
13	INC	DE
1C	INC	E
24	INC	H
23	INC	HL
DD23	INC	IX
FD23	INC	IY
2C	INC	L
33	INC	SP
DB20	IN	A,(n)

OBJ CODE	QUELLBEFEHL	
EDAA	IND	
EDBA	INDR	
EDA2	INI	
EDB2	INIR	
C38405	JP	nn
E9	JP	(HL)
DDE9	JP	(IX)
FDE9	JP	(IY)
DA8405	JP	C,nn
FA8405	JP	M,nn
D28405	JP	NC,nn
C28405	JP	NZ,nn
F28405	JP	P,nn
EA8405	JP	PE,nn
E28405	JP	PO,nn
CA8405	JP	Z,nn
382E	JR	C,e
302E	JR	NC,e
202E	JR	NZ,e
282E	JR	Z,e
182E	JR	e
02	LD	(BC),A
12	LD	(DE),A
77	LD	(HL),A
70	LD	(HL),B
71	LD	(HL),C
72	LD	(HL),D
73	LD	(HL),E
74	LD	(HL),H
75	LD	(HL),L
3620	LD	(HL),n
DD7705	LD	(IX+d),A
DD7005	LD	(IX+d),B
DD7105	LD	(IX+d),C
DD7205	LD	(IX+d),D
DD7305	LD	(IX+d),E
DD7405	LD	(IX+d),H
DD7505	LD	(IX+d),L
DD360520	LD	(IX+d),n
FD7705	LD	(IY+d),A
FD7005	LD	(IY+d),B
FD7105	LD	(IY+d),C
FD7205	LD	(IY+d),D
FD7305	LD	(IY+d),E
FD7405	LD	(IY+d),H
FD7505	LD	(IY+d),L
FD360520	LD	(IY+d),n
328405	LD	(nn),A
ED438405	LD	(nn),BC
ED538405	LD	(nn),DE
228405	LD	(nn),HL
DD228405	LD	(nn),IX
FD228405	LD	(nn),IY
ED738405	LD	(nn),SP
0A	LD	A,(BC)
1A	LD	A,(DE)
7E	LD	A,(HL)

OBJ CODE	QUELLBEFEHL	
DD7E05	LD	A,(IX+d)
FD7E05	LD	A,(IY+d)
3A8405	LD	A,(nn)
7F	LD	A,A
78	LD	A,B
79	LD	A,C
7A	LD	A,D
7B	LD	A,E
7C	LD	A,H
ED57	LD	A,I
7D	LD	A,L
3E20	LD	A,n
ED5F	LD	A,R
46	LD	B,(HL)
DD4605	LD	B,(IX+d)
FD4605	LD	B,(IY+d)
47	LD	B,A
40	LD	B,B
41	LD	B,C
42	LD	B,D
43	LD	B,E
44	LD	B,H
45	LD	B,L
0620	LD	B,n
ED4B8405	LD	BC,(nn)
018405	LD	BC,nn
4E	LD	C,(HL)
DD4E05	LD	C,(IX+d)
FD4E05	LD	C,(IY+d)
4F	LD	C,A
48	LD	C,B
49	LD	C,C
4A	LD	C,D
4B	LD	C,E
4C	LD	C,H
4D	LD	C,L
0E20	LD	C,n
56	LD	D,(HL)
DD5605	LD	D,(IX+d)
FD5605	LD	D,(IY+d)
57	LD	D,A
50	LD	D,B
51	LD	D,C
52	LD	D,D
53	LD	D,E
54	LD	D,H
55	LD	D,L
1620	LD	D,n
ED5B8405	LD	DE,(nn)
118405	LD	DE,nn
5E	LD	E,(HL)
DD5E05	LD	E,(IX+d)
FD5E05	LD	E,(IY+d)
5F	LD	E,A
58	LD	E,B
59	LD	E,C
5A	LD	E,D

OBJ CODE	QUELLBEFEHL	
5B	LD	E,E
5C	LD	E,H
5D	LD	E,L
1E20	LD	E,n
66	LD	H,(HL)
DD6605	LD	H,(IX+d)
FD6605	LD	H,(IY+d)
67	LD	H,A
60	LD	H,B
61	LD	H,C
62	LD	H,D
63	LD	H,E
64	LD	H,H
65	LD	H,L
2620	LD	H,n
2A8405	LD	HL,(nn)
218405	LD	HL,nn
ED47	LD	I,A
DD2A8405	LD	IX,(nn)
DD218405	LD	IX,nn
FD2A8405	LD	IY,(nn)
FD218405	LD	IY,nn
6E	LD	L,(HL)
DD6E05	LD	L,(IX+d)
FD6E05	LD	L,(IY+d)
6F	LD	L,A
68	LD	L,B
69	LD	L,C
6A	LD	L,D
6B	LD	L,E
6C	LD	L,H
6D	LD	L,L
2E20	LD	L,n
ED4F	LD	R,A
ED7B8405	LD	SP,(nn)
F9	LD	SP,HL
DDF9	LD	SP,IX
FDF9	LD	SP,IY
318405	LD	SP,nn
EDA8	LDD	
EDB8	LDDR	
EDA0	LDI	
EDB0	LDIR	
ED44	NEG	
00	NOP	
B6	OR	(HL)
DDB605	OR	(IX+d)
FDB605	OR	(IY+d)
B7	OR	A
B0	OR	B
B1	OR	C
B2	OR	D
B3	OR	E
B4	OR	H
B5	OR	L
F620	OR	n
EDBB	OTDR	

OBJ CODE	QUELLBEFEHL	
EDB3	OTIR	
ED79	OUT	(C),A
ED41	OUT	(C),B
ED49	OUT	(C),C
ED51	OUT	(C),D
ED59	OUT	(C),E
ED61	OUT	(C),H
ED69	OUT	(C),L
D320	OUT	(n),A
EDAB	OUTD	
EDA3	OUTI	
F1	POP	AF
C1	POP	BC
D1	POP	DE
E1	POP	HL
DDE1	POP	IX
FDE1	POP	IY
F5	PUSH	AF
C5	PUSH	BC
D5	PUSH	DE
E5	PUSH	HL
DDE5	PUSH	IX
FDE5	PUSH	IY
CB86	RES	0,(HL)
DDCB0586	RES	0,(IX+d)
FDCB0586	RES	0,(IY+d)
CB87	RES	0,A
CB80	RES	0,B
CB81	RES	0,C
CB82	RES	0,D
CB83	RES	0,E
CB84	RES	0,H
CB85	RES	0,L
CB8E	RES	1,(HL)
DDCB058E	RES	1,(IX+d)
FDCB058E	RES	1,(IY+d)
CB8F	RES	1,A
CB88	RES	1,B
CB89	RES	1,C
CB8A	RES	1,D
CB8B	RES	1,E
CB8C	RES	1,H
CB8D	RES	1,L
CB96	RES	2,(HL)
DDCB0596	RES	2,(IX+d)
FDCB0596	RES	2,(IY+d)
CB97	RES	2,A
CB90	RES	2,B
CB91	RES	2,C
CB92	RES	2,D
CB93	RES	2,E
CB94	RES	2,H
CB95	RES	2,L
CB9E	RES	3,(HL)
DDCB059E	RES	3,(IX+d)
FDCB059E	RES	3,(IY+d)

OBJ CODE	QUELLBEFEHL	
CB9F	RES	3,A
CB98	RES	3,B
CB99	RES	3,C
CB9A	RES	3,D
CB9B	RES	3,E
CB9C	RES	3,H
CB9D	RES	3,L
CBA6	RES	4,(HL)
DDCB05A6	RES	4,(IX+d)
FDCB05A6	RES	4,(IY+d)
CBA7	RES	4,A
CBA0	RES	4,B
CBA1	RES	4,C
CBA2	RES	4,D
DBA3	RES	4,E
CBA4	RES	4,H
CBA5	RES	4,L
CBAE	RES	5,(HL)
DDCB05AE	RES	5,(IX+d)
FDCB05AE	RES	5,(IY+d)
CBAF	RES	5,A
CBA8	RES	5,B
CBA9	RES	5,C
CBAA	RES	5,D
CBAB	RES	5,E
CBAC	RES	5,H
CBAD	RES	5,L
CBB6	RES	6,(HL)
DDCB05B6	RES	6,(IX+d)
FDCB05B6	RES	6,(IY+d)
CBB7	RES	6,A
CBB0	RES	6,B
CBB1	RES	6,C
CBB2	RES	6,D
CBB3	RES	6,E
CBB4	RES	6,H
CBB5	RES	6,L
CBBE	RES	7,(HL)
DDCB05BE	RES	7,(IX+d)
FDCB05BE	RES	7,(IY+d)
CBBF	RES	7,A
CBB8	RES	7,B
CBB9	RES	7,C
CBBA	RES	7,D
CBBB	RES	7,E
CBBC	RES	7,H
CBBD	RES	7,L
C9	RET	
D8	RET	C
F8	RET	M
D0	RET	NC
C0	RET	NZ
F0	RET	P
E8	RET	PE
E0	RET	PO
C8	RET	Z

OBJ CODE	QUELLBEFEHL	
ED4D	RETI	
ED45	RETN	
CB16	RL	(HL)
DDCB0516	RL	(IX+d)
FDCB0516	RL	(IY+d)
CB17	RL	A
CB10	RL	B
CB11	RL	C
CB12	RL	D
CB13	RL	E
CB14	RL	H
CB15	RL	L
17	RLA	
CB06	RLC	(HL)
DDCB0506	RLC	(IX+d)
FDCB0506	RLC	(IY+d)
CB07	RLC	A
CB00	RLC	B
CB01	RLC	C
CB02	RLC	D
CB03	RLC	E
CB04	RLC	H
CB05	RLC	L
07	RLCA	
ED6F	RLD	
CB1E	RR	(HL)
DDCB051E	RR	(IX+d)
FDCB051E	RR	(IY+d)
CB1F	RR	A
CB18	RR	B
CB19	RR	C
CB1A	RR	D
CB1B	RR	E
CB1C	RR	H
CB1D	RR	L
1F	RRA	
CB0E	RRC	(HL)
DDCB050E	RRC	(IX+d)
FDCB050E	RRC	(IY+d)
CB0F	RRC	A
CB08	RRC	B
CB09	RRC	C
CB0A	RRC	D
CB0B	RRC	E
CB0C	RRC	H
CB0D	RRC	L
OF	RRCA	
ED67	RRD	
C7	RST	00H
CF	RST	08H
D7	RST	10H
DF	RST	18H
E7	RST	20H
EF	RST	28H
F7	RST	30H
FF	RST	38H
DE20	SBC	A,n

OBJ CODE	QUELLBEFEHL	
9E	SBC	A,(HL)
DD9E05	SBC	A,(IX+d)
FD9E05	SBC	A,(IY+d)
9F	SBC	A,A
98	SBC	A,B
99	SBC	A,C
9A	SBC	A,D
9B	SBC	A,E
9C	SBC	A,H
9D	SBC	A,L
ED42	SBC	HL,BC
ED52	SBC	HL,DE
ED62	SBC	HL,HL
ED72	SBC	HL,SP
37	SCF	
CBC6	SET	0,(HL)
DDCB05C6	SET	0,(IX+d)
FDCB05C6	SET	0,(IY+d)
CBC7	SET	0,A
CBC0	SET	0,B
CBC1	SET	0,C
CBC2	SET	0,D
CBC3	SET	0,E
CBC4	SET	0,H
CBC5	SET	0,L
CBCE	SET	1,(HL)
DDCB05CE	SET	1,(IX+d)
FDCB05CE	SET	1,(IY+d)
CBCF	SET	1,A
CBC8	SET	1,B
CBC9	SET	1,C
CBCA	SET	1,D
CBCB	SET	1,E
CBCC	SET	1,H
CBCD	SET	1,L
CBD6	SET	2,(HL)
DDCB05D6	SET	2,(IX+d)
FDCB05D6	SET	2,(IY+d)
CBD7	SET	2,A
CBD0	SET	2,B
CBD1	SET	2,C
CBD2	SET	2,D
CBD3	SET	2,E
CBD4	SET	2,H
CBD5	SET	2,L
CBD8	SET	3,B
CBDE	SET	3,(HL)
DDCB05DE	SET	3,(IX+d)
FDCB05DE	SET	3,(IY+d)
CBDF	SET	3,A
CBD9	SET	3,C
CBDA	SET	3,D
CBDB	SET	3,E
CBDC	SET	3,H
CBDD	SET	3,L
CBE6	SET	4,(HL)

OBJ CODE	QUELLBEFEHL	
DDCB05E6	SET	4,(IX+d)
FDCB05E6	SET	4,(IY+d)
CBE7	SET	4,A
CBE0	SET	4,B
CBE1	SET	4,C
CBE2	SET	4,D
CBE3	SET	4,E
CBE4	SET	4,H
CBE5	SET	4,L
CBEE	SET	5,(HL)
DDCB05EE	SET	5,(IX+d)
FDCB05EE	SET	5,(IY+d)
CBEF	SET	5,A
CBE8	SET	5,B
CBE9	SET	5,C
CBEA	SET	5,D
CBEB	SET	5,E
CBEC	SET	5,H
CBED	SET	5,L
CBF6	SET	6,(HL)
DDCB05F6	SET	6,(IX+d)
FDCB05F6	SET	6,(IY+d)
CBF7	SET	6,A
CBF0	SET	6,B
CBF1	SET	6,C
CBF2	SET	6,D
CBF3	SET	6,E
CBF4	SET	6,H
CBF5	SET	6,L
CBFE	SET	7,(HL)
DDCB05FE	SET	7,(IX+d)
FDCB05FE	SET	7,(IY+d)
CBFF	SET	7,A
CBF8	SET	7,B
CBF9	SET	7,C
CBFA	SET	7,D
CBFB	SET	7,E
CBFC	SET	7,H
CBFD	SET	7,L
CB26	SLA	(HL)
DDCB0526	SLA	(IX+d)
FDCB0526	SLA	(IY+d)
CB27	SLA	A
CB20	SLA	B
CB21	SLA	C
CB22	SLA	D
CB23	SLA	E
CB24	SLA	H
CB25	SLA	L
CB2E	SRA	(HL)
DDCB052E	SRA	(IX+d)
FDCB052E	SRA	(IY+d)
CB2F	SRA	A
CB28	SRA	B
CB29	SRA	C
CB2A	SRA	D

OBJ CODE	QUELLBEFEHL	
CB2B	SRA	E
CB2C	SRA	H
CB2D	SRA	L
CB3E	SRL	(HL)
DDCB053E	SRL	(IX+d)
FDCB053E	SRL	(IY+d)
CB3F	SRL	A
CB38	SRL	B
CB39	SRL	C
CB3A	SRL	D
CB3B	SRL	E
CB3C	SRL	H
CB3D	SRL	L
96	SUB	(HL)
DD9605	SUB	(IX+d)
FD9605	SUB	(IY+d)
97	SUB	A
90	SUB	B
91	SUB	C
92	SUB	D
93	SUB	E
94	SUB	H
95	SUB	L
D620	SUB	n
AE	XOR	(HL)
DDAE05	XOR	(IX+d)
FDAE05	XOR	(IY+d)
AF	XOR	A
A8	XOR	B
A9	XOR	C
AA	XOR	D
AB	XOR	E
AC	XOR	H
AD	XOR	L
EE20	XOR	n

(Courtesy of Zilog Inc.)

Anhang F

Äquivalente Befehle: Z80–8080

Z80	8080	Z80	8080	Z80	8080
ADC A, (HL)	ADC M	EX (SP), HL	XTHL	OR n	ORI [B2]
ADC A, n	ACI [B2]	HALT	HLT	OR r	ORA r
ADC A, r	ADC r	IN A, (n)	IN [B2]	OR (HL)	ORA M
ADD A, (HL)	ADD M	INC BC	INX B	OUT (n), A	OUT [B2]
ADD A, n	ADI [B2]	INC DE	INX D	POP AF	POP PSW
ADD A, r	ADD r	INC HL	INX H	POP BC	POP B
ADD HL, BC	DAD B	INC r	INR r	POP DE	POP D
ADD HL, DE	DAD D	INC SP	INX SP	POP HL	POP H
ADD HL, HL	DAD H	INC (HL)	INR M	PUSH AF	PUSH PSW
ADD HL, SP	DAD SP	JP C, nn	JC [B2] [B3]	PUSH BC	PUSH B
AND n	ANI [B2]	JP M, nn	JM [B2][B3]	PUSH DE	PUSH D
AND r	ANA r	JP NC, nn	JNC [B2] [B3]	PUSH HL	PUSH H
AND (HL)	ANA M	JP nn	JMP [B2] [B3]	RET	RET
CALL C, nn	CC [B2] [B3]	JP NZ, nn	JNZ [B2] [B3]	RET C	RC
CALL M, nn	CM [B2] [B3]	JP P, nn	JP [B2] [B3]	RET M	RM
CALL NC, nn	CNC [B2] [B3]	JP PE, nn	JPE [B2][B3]	RET NC	RNC
CALL nn	CALL	JP PO, nn	JPO [B2][B3]	RET NZ	RNZ
CALL NZ, nn	CNZ [B2] [B3]	JP Z, nn	JZ [B2] [B3]	RET P	RP
CALL P, nn	CP [B2] [B3]	JP (HL)	PCHL	RET PE	RPE
CALL PE, nn	CPE [B2] [B3]	LD A, (DE)	LDAX	RET PO	RPO
CALL PO, nn	CPO [B2] [B3]	LDA, (nn)	LDA [B2] [B3]	RET Z	RZ
CALL Z, nn	CZ [B2] [B3]	LD DE, nn	LXID, [B2] [B3]	RLA	RAL
CCF	CMC	LD SP, nn	LXI SP, [B2] [B3]	RLCA	RLC
CP r	CMP r	LD (BC), A	STAX B	RRA	RAR
CP (HL)	CMP M	LD (DE), A	STAX D	RRCA	RRC
CPL	CMA	LD (HL), r	MOV M, r	RST P	RST P
CP n	CPI [B2]	LD (nn), A	STA [B2] [B3]	SBC A, (HL)	SBB M
DAA	DAA	LD (nn), HL	SHLD [B2] [B3]	SBC A, n	SBI [B2]
DEC BC	DCX B	LD A, (BC)	LDAX B	SBC A, r	SBB r
DEC DE	DCX D	LD BC, nn	LXIB, [B2] [B3]	SCF	STC
DEC HL	DCX H	LD HL, (nn)	LHLD [B2] [B3]	SUB n	SUI [B2]
DEC r	DCR r	LD HL, nn	LXI H [B2] [B3]	SUB r	SUB r
DEC SP	DCX SP	LD r, (HC)	MOV 1, M	SUB (HL)	SUB M
DEC (HL)	DCR M	LD r, n	MVI r, [B2]	XOR n	XRI [B2]
DI	DI	LD r, r'	MOV r1, r2	XOR r	XRA r
EI	EI	LD SP, HL	SPHL	XOR (HL)	XRA M
EX DE, HL	XCHG	NOP	NOP		

Anhang G

Äquivalente Befehle: 8080–Z80

8080	Z80	8080	Z80	8080	Z80
ACI [B2]	ADC A, n	IN [B2]	IN A, (n)	POP H	POP HL
ADC M	ADC A, (HL)	INR M	INC (HL)	POP PSW	POP AF
ADC r	ADC A, r	INR r	INC r	PUSH B	PUSH BC
ADD M	ADD A, (HL)	INX B	INC BC	PUSH D	PUSH DE
ADD r	ADD A, r	INX D	INC DE	PUSH H	PUSH HL
ADI [B2]	ADD A, n	INX H	INC HL	PUSH PSW	PUSH AF
ANA M	AND (HL)	INX SP	INC SP	RAL	RLA
ANA r	AND r	JC [B2] [B3]	JP C, nn	RAR	RRA
ANI [B2]	AND n	JM [B2] [B3]	JP M, nn	RC	RET C
CALL	CALL nn	JMP [B2] [B3]	JP nn	RET	RET
CC [B2] [B3]	CALL C, nn	JNC [B2] [B3]	JP NC, nn	RLC	RLCA
CM [B2] [B3]	CALL M, nn	JNZ [B2] [B3]	JP NZ, nn	RM	RET M
CMA	CPL	JP [B2] [B3]	JP P, nn	RNC	RET NC
CMC	CCF	JPE [B2] [B3]	JP PE, nn	RNZ	RET NZ
CMP M	CP (HL)	JPO [B2] [B3]	JP PO, nn	RP	RET P
CMP r	CP r	JZ [B2] [B3]	JP Z, nn	RPE	RET PE
CNC [B2] [B3]	CALL NC, nn	LDA [B2] [B3]	LD A, (nn)	RPO	RET PO
CNZ [B2] [B3]	CALL NZ, nn	LDAX B	LD A, (BC)	RRC	RRCA
CP [B2] [B3]	CALL P, nn	LDAX D	LD A, (DE)	RST	RST P
CPE [B2] [B3]	CALL PE, nn	LXI D [B2] [B3]	LD HL, (nn)	RZ	RET Z
CPI [B2]	CP n	LXI B [B2] [B3]	LD BC, nn	SBB M	SBC A, (HL)
CPO [B2] [B3]	CALL PO, nn	LDID [B2] [B3]	LD DE, nn	SBB r	SBC A, r
CZ [B2] [B3]	CALL Z, nn	LXI H [B2] [B3]	LD HL, nn	SBI [B2]	SBC A, n
DAA	DAA	LXI SP [B2] [B3]	LD SP, nn	SHLD [B2] [B3]	LD (nn), HL
DAD B	ADD HL, BC	MOV M, r	LD (HL), r	SPHL	LD SP, HL
DAD D	ADD HL, DE	MOV r, M	LD r, (HL)	STA [B2] [B3]	LD (nn), A
DAD H	ADD HL, HL	MOV r1, r2	LD r, r'	STAX B	LD (BC), A
DAD SP	ADD HL, SP	MVI M	LD (HL), n	STAX D	LD (DE), A
DCR M	DEC (HL)	MVI r [B2]	LD r, n	STC	SCF
DCR r	DEC r	NOP	NOP	SUB M	SUB (HL)
DCX B	DEC BC	ORA M	OR (HL)	SUB r	SUB r
DCX D	DEC DE	ORA r	OR r	SUI [B2]	SUB n
DCX H	DEC HL	ORI [B2]	OR n	XCHG	EX DE, HL
DCX SP	DEC SP	OUT [B2]	OUT (n), A	XRA M	XOR (HL)
DI	DI	PCHL	JP (HL)	XRA r	XOR r
EI	EI	POP B	POP BC	XRI [B2]	XOR n
HALT	HLT	POP D	POP DE	XTHL	EX (SP), HL

Index

A

absolute Adressierung 102, 431, 436
ADC, A, s 180
ADC HL, ss 182
ADD A, (HL) 184
ADD A, (IX + d) 186
ADD A, (IY + d) 188
ADD A, n 190
ADD A, r 191
ADD HL, ss 193
ADD IX, rr 195
ADD IY, rr 197
Addition 90, 94, 104, 445
Adreßbus 44
Adressierungstechniken 429
Adreßregister 47, 48
Akkumulator 46
Algorithmus 17, 18
alphabetische Liste 543
ALU 43, 46, 57
AND s 199
AND 159
APL 565
Arithmetik 157
Arithmetik-Logik-Einheit 43
Arithmetikprogramme 90
arithmetische Befehle 106
ASCII 37, 472
ASCII-Tabelle 38
Assembler 566, 573
Assemblerdirektive 579
Assemblerdirektiven 138
Assembler-Kommando 580
Assembler-Liste 574
Assembler-Programm 90
Assemblersprache 63, 564, 576
asynchrone Übertragung 459
Ausdruck 579
Ausführen 67
Ausgabe 449
Ausgabebefehl 175
Austauschbefehl 154

B

BASIC 565
Baudot-Kode 472
Baum 530
BCD 34
BCD-Addition 104
BCD-Arithmetik 101
BCD-Subtraktion 104
Bearbeitung von Daten 148
bedingte Assemblierung 582
bedingter Befehl 47
Befehl 18, 90
Befehlsfeld 574
Befehlsregister 51, 60
Befehlstypen 106
Befehlszähler 49
Benchmark 458
Betriebssystem 566
binär 39
Binärziffer 19
BIT b, (HL) 201
BIT b, (IX + d) 203
BIT b, (IY + d) 205
BIT b, r 207
Bitadressierung 438
Bitverarbeitung 163
Block 529
Blocksuchbefehl 156
Blocktransfer 440
Blocktransferbefehl 155
Bootstrap 45

Breakpoint 567
Break-Zeichen 457
Bubble-Sort 519
Bus 44
BUSAK 86
Busrequest 483
BUSRQ 86, 483
Byte 20

C
C 165
CALL cc, pq 209
CALL pq 212
Carry 165
CCF 214
chronologische Struktur 50
Codeumwandlung 511
Compiler 565, 566
CP s 215
CPD 217
CPDR 219
CPI 221
CPIR 223
CPL 225
CPU 43
CRC 504
Cross-Assembler 572

D
DAA 226
Datenbus 44
Datenstrukturen 525
Datenübertragung 106
Datenverarbeitung 157
Debugger 567
Debugging 18
DEC m 228
DEC rr 230
DEC IX 232
DEC IY 233
Decode 51
Dekoder 60
Dekodieren 67
Dekodierlogik 45
Dekrementierbefehl 113
DI 234
Dienstprogramme 567
direkte Adressierung 432
direkte Indizierung 432
direkter Befehl 83
Direktiven 564
Division 125, 127, 129
DJNZ e 235
DMA 86, 478, 483
Dokumentation 18
DOS 566
duale Form 20
Dualsystem 20

E
EBCDIC 37
Editor 566
EI 237
Ein-/Ausgabebefehl 173
Ein-/Ausgabegeräte 497
Ein-/Ausgabetechniken 449
Einerkomplement 25
Einfügen 536, 547, 556
Eingabe 449
Eingabebefehl 176
Emulator 567
Entwicklungssystem 571
erweiterte Adressierung 153, 432, 436
EX AF, AF' 238
EX DE, HL 239
EX (SP), HL 240
EX (SP), IX 242
EX (SP), IY 244
Execute 51
exklusives OR 160
Exponent 36
EXX 246

F
Fehlermeldung 574
Fernschreiber 472
Fetch 51
Flag 30, 47, 169
Flag-Register 57

G
ganze Zahl 20
gepackte BCD 34, 101
getaktete Logik 81
Gleitkomma 36

H
H 167
Halbübertrag 167
HALT 86, 247

Haltepunkt 567
Handshaking 465
Hardwarestapel 50
Hardwareverzögerung 454
HDLC 504
hexadezimal 39
Hexadezimal-Tabelle 41
hexadezimale Kodierung 563
Hilfsprogramme 567

I
IBM-BSC 504
IM0 248
IM1 249
IM2 250
implizierte Adressierung 435
implizite Adressierung 429
impliziter Befehl 70
Impuls 451
IN r, (C) 251
IN A, (N) 253
In-Circuit-Emulator 569
INC r 254
INC rr 255
INC (HL) 257
INC (IX + d) 258
INC (IY + d) 260
INC IX 262
INC IY 263
IND 264
Indexregister 49
indirekte Indizierung 432
indirekte Adressierung 438
Indizieren 59
indizierte Adressierung 153, 432, 437
INDR 266
Inhaltsverzeichnis 527
INI 268
INIR 270
Initialisierung 109
Instruction Register 51
INT 86
INTA 487
Interfacebaustein 45
Interpreter 565, 566
Interpretieren 64
Interrupt 465, 480, 486
Interrupt Acknowledge 487
Interruptregister 59
IORQ 86

J
JP cc, pq 272
JP pq 274
JP (HL) 275
JP (IX) 276
JP (IY) 277
JR cc, e 278
JR e 280

K
Kodierung 18
Kombination der Adressierungsarten 435, 438
Kombinationsbaustein 45
Kommentarfeld 90, 574
Konstante 577
Kontrollbus 44
Kurzadressierung 431
Kurzbefehl 20
kurze Adressierung 439

L
Ladebefehl 91
lange Adressierung 439
LD dd, (nn) 281
LD dd, nn 283
LD r, n 285
LD r, r' 287
LD (BC), A 289
LD (DE), A 290
LD (HL), n 291
LD (HL), r 293
LD r, (IX + d) 295
LD r, (IY + d) 297
LD (IX + d), n 299
LD (IY + d), n 301
LD (IX + d), r 303
LD (IY + d), r 305
LD A, (nn) 307
LD (nn), A 309
LD (nn), dd 311
LD (nn), HL 313
LD (nn), IX 315
LD (nn), IY 317
LD A, (BC) 319
LD A, (DE) 320
LD A, I 321
LD I, A 322
LD A, R 323
LD HL, (nn) 324

LD IX, nn 326
LD IX, (nn) 328
LD IY, nn 330
LD IY, (nn) 332
LD R, A 334
LD r, (HL) 335
LD SP, HL 337
LD SP, IX 338
LD SP, IY 339
LDD 340
LDDR 342
LDI 344
LDIR 346
LED 468
Lesespeicher 44
LIFO 50
Liste 526, 535
Literal 431, 577
Loader 567
Löschen 537, 547, 558
Logik 158
logische Operationen 132
logische Befehle 132

M
M1 87
Makro 580
Makroparameter 581
Mantisse 36
Markenfeld 574
mehrfache Genauigkeit 94
Memory Mapped I/O 150
Mikrobefehl 81
mnemotechnisch 63
Monitor 45, 566
MPU 43
MREQ 87
Multiplikation 106, 108, 123

N
N 166
natürliche Sprache 17
NEG 348
Nibble 20
nicht wiederherstellendes Verfahren 126
nicht wiederherstellende Division
nichtmaskierbarer Interrupt 485
NMI 86, 485
NOP 349
normalisiert 36
Null 168

O
Objektkode 566
oktal 39
Operator 578
OR s 350
OR 159
OTOR 352
OTIR 354
OUT (C), r 356
OUT (N), A 358
OUTD 359
OUTI 361

P
P/V 166
Parallel-Ein-/Ausgabebaustein 45
parallele Übertragung 456
Parität 166
Paritätsbit 511
PASCAL 565
PC 49
Peripherie 478
PIO 45, 497
Polling 478
POP qq 363
POP IX 365
POP IY 367
Pop 50
Port 501
Programm Counter 49
Programm 18
Programmierebenen 565
Programmiersprache 17
Programmierung 18
Programmschleife 113
Prüfsumme 515
Pseudobefehl 94
Pufferregister 57
Pull 50
Pulsdauer 455
PUSH qq 369
PUSH IX 371
PUSH IY 373
Push 50

Q
Quellregister 63

R
RAM 45
Random-Access-Memory 45
Raute 108
RD 87
Refreshregister 59
Registeradressierung 429
Rekursion 140
relative Adressierung 432, 437
relativer Sprung 149
RES b, s 375
RESET 87
Restart 173
RET 378
RET cc 380
RETI 382
RETN 384
RFSH 87
Richtungsregister 501
Ringliste 530
RL s 386
RLA 388
RLC r 390
RLC (HL) 392
RLC (IX + d) 394
RLC (IY/+ d) 396
RLCA 389
RLD 398
ROM 44
Rotieren 46, 160
RR s 400
RRA 402
RRC s 403
RRCA 405
RRD 406
RST p 408
RST 173
Rubout 457
Rücksprung 134

S
S 168
SBC A, s 410
SBC HL, ss 412
SCF 414
Schieben 46, 160
Schlange 528
Schleife 59
Schreib-/Lesespeicher 45
SDLC 504
Seite-Null-Adressierung 431, 436
sequentielle Liste 526
serielle Übertragung 459
SET b, s 415
Sign 168
Signal 450
Simulator 567
SLA s 418
Softwarestapel 50
Sortieren 531
SP 49
Speicherzyklus 51
Sprung 83, 149, 163
Sprungbefehl 171
SRA s 420
SRL s 422
Stapel 50, 529
Stapelzeiger 49
Startbit 472
Status 80
Statusbit 461
Statusinformation 498
Statusregister 47
Steuerbefehl 150, 176
Steuerregister 498
Steuerwerk 60
Stopbit 472
Stromschleife 472
SUB s 424
Subtraktion 99, 104, 166
Suchen 531, 536, 543, 556
Symbol 576
symbolisch 63
symbolische Adresse 93
synchrone Übertragung 459
Systemarchitektur 43

T
Tabelle 512, 574
Taktbit 461
Taktgeber 44
Test 149, 163, 510
Time-Sharing 572
Timer 454
Trace 569
Transfer 147, 151
Transferbefehl 147
Treiber 45
TTY 473

U
UART 465
Überlappungstechnik 73
Überlauf 29, 166
Übertrag 29, 165
Überziehung 125
Universalregister 47
unmittelbare Operation 64
unmittelbare Adressierung 102, 152, 431, 436
Unterprogramm 581
Unterprogrammaufruf 134
Unterprogrammbibliothek 141
Unterprogramme 134
Unterprogrammparameter 140
unzulässiger Kode 101
Urlader 45
USRT 504
Utility Routines 567

V
Vektorinterrupt 490
Vergleich 517
verkettete Liste 527, 529
verschachtelte Aufrufe 137
Verzögerung 452
Verzweigung 18, 83, 106, 108
Vorzeichen 168
vorzeichenbehaftete Darstellung 24

W
WAIT 86
wiederherstellendes Verfahren 126
WR 87

X
XOR s 426
XOR 159

Z
Z 168
Z80-PIO 502
Z80-SIO 504
Zähler 48, 454
Zehnerkomplement 103
Zeiger 48, 525
Zeitgeber 454
Zero 168
Zielregister 63
Zweierkomplement 26
Zwischenspeicher 57

BASIC FÜR DEN KAUFMANN
von D. Hergert – das BASIC-Buch für Studenten und Praktiker im kaufmännischen Bereich. Enthält Anwendungsbeispiele für Verkaufs- und Finanzberichte, Grafiken, Abschreibungen u.v.m. 208 Seiten, 85 Abbildungen, Ref.-Nr.: **3026** (1983)

SINCLAIR ZX SPECTRUM BASIC HANDBUCH
von D. Hergert – eine wichtige Hilfe für jeden SPECTRUM-Anwender. Gibt eine Übersicht aller BASIC-Begriffe, die auf diesem Rechner verwendet werden können, und erläutert sie ausführlich anhand von Beispielen. 288 Seiten, 188 Abbildungen, Ref.-Nr.: **3027** (1983)

SINCLAIR ZX81 BASIC HANDBUCH
von D. Hergert – vermittelt Ihnen das vollständige BASIC-Vokabular anhand von praktischen Beispielen, macht Sie zum Programmierer Ihres ZX81. 181 Seiten, 120 Abbildungen, Ref.-Nr.: **3028** (1983)

ERFOLG MIT VisiCalc
von D. Hergert – umfassende Einführung in VisiCalc und seine Anwendung. Zeigt Ihnen u. a.: Aufstellung eines Verteilungsbogens, Benutzung von VisiCalc-Formeln, Verwendung der DIF-Datei-Funktion. 224 Seiten, 58 Abbildungen, Ref.-Nr.: **3030** (1983)

IBM PC-DOS HANDBUCH
von R. A. King – umfassende Einführung in das Disketten-Betriebssystem Ihres IBM PC, seine grundsätzlichen Möglichkeiten und Funktionen sowie auch fortgeschrittene Funktionen (einschließlich der Version 2.0). Ca. 320 Seiten, ca. 50 Abbildungen, Ref.-Nr.: **3034** (Mai 1984)

APPLE II BASIC HANDBUCH
von D. Hergert – ein handliches Nachschlagewerk, das neben Ihren Apple II, II+ oder IIe stehen sollte. Dank vieler Tips und Vorschläge eine wesentliche Erleichterung fürs Programmieren. Ca. 272 Seiten, 116 Abbildungen, Ref.-Nr. **3036** (April 1984)

Z80 ANWENDUNGEN
von J. W. Coffron – vermittelt alle nötigen Anweisungen, um Peripherie-Bausteine mit dem Z80 zu steuern und individuelle Hardware-Lösungen zu realisieren. Ca. 320 Seiten, ca. 200 Abbildungen, Ref.-Nr.: **3037** (Juni 1984)

CP/M-HANDBUCH
von Rodnay Zaks – das Standardwerk über CP/M, das meistgebrauchte Betriebssystem für Mikrocomputer. Für Anfänger eine verständliche Einführung, für Fortgeschrittene ein umfassendes Nachschlagewerk über die CP/M-Versionen 2.2, 3.0 und CCP/M-86 sowie MP/M. 2., überarbeitete Ausgabe. Ca. 350 Seiten, ca. 100 Abbildungen, Ref.-Nr.: **3053** (1984)

FORDERN SIE EIN GESAMTVERZEICHNIS UNSERER VERLAGSPRODUKTION AN:

SYBEX-VERLAG GmbH
Vogelsanger Weg 111
4000 Düsseldorf 30
Tel.: (02 11) 62 64 41
Telex: 8 588 163

SYBEX
6–8, Impasse du Curé
75018 Paris
Tel.: 1/203-95-95
Telex: 211.801 f

SYBEX INC.
2344 Sixth Street
Berkeley, CA 94710, USA
Tel.: (415) 848-8233
Telex: 336311

Die SYBEX Bibliothek

EINFÜHRUNG IN PASCAL UND UCSD/PASCAL
von Rodnay Zaks – das Buch für jeden, der die Programmiersprache PASCAL lernen möchte. Vorkenntnisse in Computerprogrammierung werden nicht vorausgesetzt. Eine schrittweise Einführung mit vielen Übungen und Beispielen. 535 Seiten, 130 Abbildungen, Ref.Nr.: **3004** (1982)

DAS PASCAL HANDBUCH
von Jacques Tiberghien – ein Wörterbuch mit jeder Pascal-Anweisung und jedem Symbol, reservierten Wort, Bezeichner und Operator, für beinahe alle bekannten Pascal-Versionen. 480 Seiten, 270 Abbildungen, Format 23 x 18 cm, Ref.Nr.: **3005** (1982)

PASCAL PROGRAMME – MATHEMATIK, STATISTIK, INFORMATIK
von Alan Miller – eine Sammlung von 60 der wichtigsten wissenschaftlichen Algorithmen samt Programmauflistung und Musterdurchlauf. Ein wichtiges Hilfsmittel für Pascal-Benutzer mit technischen Anwendungen. 398 Seiten, 120 Abbildungen, Format 23 x 18 cm, Ref.Nr.: **3007** (1982)

POCKET MIKROCOMPUTER LEXIKON
– die schnelle Informations-Börse! 1300 Definitionen, Kurzformeln, technische Daten, Lieferanten-Adressen, ein englisch-deutsches und französisch-deutsches Wörterbuch. 176 Seiten, Format DIN A6, Ref.Nr. **3008** (1982)

PROGRAMMIERUNG DES 6502 (2. überarbeitete Ausgabe)
von Rodnay Zaks – Programmierung in Maschinensprache mit dem Mikroprozessor 6502, von den Grundkonzepten bis hin zu fortgeschrittenen Informationsstrukturen. 368 Seiten, 160 Abbildungen, Format DIN A5, Ref.Nr.: **3011** (1982)

MIKROPROZESSOR INTERFACE TECHNIKEN (3. überarbeitete Ausgabe)
von Rodnay Zaks/Austin Lesea – Hardware- und Software-Verbindungstechniken samt Digital/Analog-Wandler, Peripheriegeräte, Standard-Busse und Fehlersuchtechniken. 432 Seiten, 400 Abbildungen, Format DIN A5, Ref.Nr.: **3012** (1982)

6502 ANWENDUNGEN
von Rodnay Zaks – das Eingabe-/Ausgabe-Buch für Ihren 6502-Microprozessor. Stellt die meistgenutzten Programme und die dafür notwendigen Hardware-Komponenten vor. 288 Seiten, 213 Abbildungen, Ref.Nr.: **3014** (1983)

BASIC PROGRAMME – MATHEMATIK, STATISTIK, INFORMATIK
von Alan Miller – eine Bibliothek von Programmen zu den wichtigsten Problemlösungen mit numerischen Verfahren, alle in BASIC geschrieben, mit Musterlauf und Programmlisting. 352 Seiten, 147 Abbildungen, Ref.Nr.: **3015** (1983)

CHIP UND SYSTEM: Einführung in die Mikroprozessoren-Technik
von Rodnay Zaks – eine sehr gut lesbare Einführung in die faszinierende Welt der Computer, vom Microprozessor bis hin zum vollständigen System. 576 Seiten, 325 Abbildungen, Ref.Nr.: **3017** (1984)

EINFÜHRUNG IN WORDSTAR
von Arthur Naiman – eine klar gegliederte Einführung, die aufzeigt, wie das Textbearbeitungsprogramm WORDSTAR funktioniert, was man damit tun kann und wie es eingesetzt wird. 240 Seiten, 36 Abbildungen, Ref.Nr.: **3019** (1983)

PLANEN UND ENTSCHEIDEN MIT BASIC
von X. T. Bui – eine Sammlung von interaktiven, kommerziell-orientierten BASIC-Programmen für Management- und Planungsentscheidungen. 200 Seiten, 53 Abbildungen, Ref.-Nr.: **3025** (1983)

Über das Buch:

Das Buch, PROGRAMMIERUNG DES Z 80, ist als Lehr- und Lernmittel konzipiert und entworfen worden. Es ist außerdem ein umfassendes Nachschlagewerk für diesen Mikroprozessor und eine gründliche Einführung in die Programmierung mit Maschinensprachen.

Ausgehend von den grundlegenden Konzepten bis hin zu fortgeschrittenen Datenstrukturen und Techniken, zeigt Ihnen dieses Buch anhand von vielen Abbildungen und zahlreichen Beispielen, wie Sie gut organisierte Programme in der Sprache des Z 80 schreiben können. Alle Konzepte werden einfach und präzise beschrieben, um dann zum Aufbau schwieriger Techniken benutzt zu werden. Mit diesem Buch lernen Sie nicht nur die Programmierungstechniken, Sie erhalten auch ein detailliertes Verständnis dafür, wie Mikroprozessoren die Instruktionen eigentlich ausführen.

Folgende Themen werden in diesem Buch ausführlich behandelt:

- Z 80-Hardware-Organisation
- Input/Output-Techniken
- Der komplette Befehlssatz
- Z 80 Adressierungsmodi
- Theorie und Entwurf von Datenstrukturen
- Anwendungsbeispiele

Mit über 200 Abbildungen, einem gründlichen Stichwortverzeichnis und technischen Informationen in sieben Anhängen, ist dies ein unentbehrliches Buch für Techniker, Studenten und jeden, der an der Programmierung mit Maschinensprache interessiert ist.

Über den Autor:

Rodnay Zaks erhielt seinen Doktor in Informatik von der University of California, Berkeley, entwickelte ein mikroprogrammiertes APL-System, arbeitet im „Silicon-Valley", wo er ein Wegbereiter für die industrielle Anwendung von Mikroprozessoren war, und gab jahrelang Unterricht über Programmierung und Entwicklung sowie Anwendung von Mikroprozessoren. Seine Bücher reflektieren diese Erfahrung – sie sind technisch präzise, didaktisch strukturiert und praktisch orientiert. Der sehr klare Schreibstil führt zu einer maximalen Informations-Wiedergabe und -Aufnahme.

Die Bücher von Dr. Rodnay Zaks sind inzwischen in mehr als zehn Sprachen übersetzt worden. Viele seiner Bücher sind zu Standardwerken geworden und einige davon sind Dauer-Bestseller.